The QPE User's Manual
(QPE Version 2.6)

Kenneth D. Forbus

Qualitative Reasoning Group
The Institute for the Learning Sciences
Northwestern University

DRAFT of January 29, 1992

Abstract

This paper describes how to use the *Qualitative Process Engine* (`QPE`), an envisioner for Qualitative Process theory. We begin with an overview of `QPE` and how to install it at your site. Next we describe how to operate it and examine the results, including comments on the various parameters and switches available. `QPE` is designed to be used as a module in other reasoning systems, so methods for interfacing it to other programs are described. A brief tutorial for domain modeling is next, including a complete listing of the primitives supplied for building domain models, and how the modeling language can be extended. Finally, we discuss plans for future releases.

Contents

1 Introduction **1**
1.1 An overview of QPE . . . 1
1.2 Conventions . . . 2
1.3 Legal Restrictions . . . 4
1.4 Feedback . . . 5

2 Installing QPE at your site **6**
2.1 Unwrapping the package . . . 6
2.2 Compiling and loading QPE . . . 7
2.3 Installing machine-specific graphical systems . . . 8
2.4 Testing QPE . . . 8

3 Operating QPE **9**
3.1 Inputs . . . 9
3.2 The generic command interface . . . 10
3.3 Interrogatives . . . 10
3.3.1 ATMoSphere interrogatives . . . 10
3.3.2 Properties of the expanded domain model . . . 11
3.3.3 Global properties of the envisionment . . . 12
3.3.4 Displaying information about quantities . . . 12
3.3.5 Displaying information about situations and sclasses . . . 12
3.3.6 Displaying information about limit analysis . . . 13
3.4 Modes . . . 13
3.4.1 Overall properties . . . 13
3.4.2 Generating View and Process structures . . . 14
3.4.3 Influence resolution . . . 14
3.4.4 Limit analysis . . . 14
3.4.5 The QPE Flight Recorder . . . 15
3.4.6 Report Generation . . . 16
3.5 Flags . . . 16
3.5.1 Available flags . . . 17
3.6 Examining the output . . . 18
3.6.1 Report Generation . . . 18
3.7 The benchmarking system . . . 19
3.8 The Event Scheduler . . . 19

4 Building Domain models **21**
4.1 An overview of the modeling process . . . 21
4.2 Built-in primitives for domain model specification . . . 21
4.2.1 Declarations . . . 21
4.2.2 Defining domain-specific choices . . . 22
4.2.3 ATMoSphere connections . . . 22
4.2.4 Connectives . . . 23
4.2.5 Basic QP primitives . . . 23
4.2.6 Existence . . . 25
4.2.7 Defining domain-specific sets . . . 25

4.3 Adding new primitives 25

5 Tips, tricks, and traps in using QPE **26**
5.1 Getting off the ground 26
5.2 Model building problems 26
5.3 When the envisionment gets too large 27
5.4 When the envisionment isn't large enough 28

6 Using QPE as a module **29**
6.1 Internal representations 29
6.2 Interface procedures 30

7 Plans for future releases **32**

References **33**

A QPE files **35**

Index **37**

1 Introduction

Simulation plays a central role in qualitative physics. Qualitative simulation provides a predictive capacity that can extract conclusions from very weak information. Traditional numerical simulations provide highly detailed answers, but they require detailed mathematical models and detailed input data. Qualitative simulation provides precise but ambiguous answers, in that it identifies the broad classes of behaviors possible given a weakly specified scenario description and domain theory. For many questions knowing these broad classes suffice, and when more detailed answers are needed the results of qualitative simulation can be used as a framework to orchestrate the application of more specific knowledge.

A particular type of qualitative simulation is *envisioning*. Theoretical descriptions of envisioning can be found in [5,10,12]. This manual describes a particular envisioner for Qualitative Process theory [8,10]. Versions of this program, the *Qualitative Process Engine* (hereafter `QPE`), have been used by our group both as a stand-alone system [11] and as a module in other qualitative reasoning systems [18,1,6] for several years now. Our intent in making `QPE` publically available is to help other researchers get started in qualitative reasoning.

The rest of this section provides a brief overview of how `QPE` works, the conventions used in the manual, and some legal stuff. `QPE` is written in Common Lisp, and signficant effort has been spent to make it portable. Section 2 describes how to install `QPE` at your site, and outlines what lisp implementations it has been tested on to date. Section 3 describes how to run `QPE`, including how to specify scenarios and examine the results. What you need to know to use `QPE` as part of a larger system is summarized in Section 6. Section 4 describes the primitives available for building domain models and how this vocabulary can be extended. While the methodology of qualitative modeling is still undergoing rapid refinement, this section also includes a brief overview of our approach and distills some of the lessons we learned the hard way.

1.1 An overview of `QPE`

`QPE` takes as input a *domain model* and a *scenario*, and produces a total envisionment. The domain model is specified in a modeling language based on QP theory, and provides general descriptions useful in a wide variety of examples. The scenario specifies both the structural description of the example, such as what objects exist and what relationships hold between them, and any modeling assumptions which hold for the particular analysis. Another paper [13] outlines how this is accomplished, and understanding it will be helpful in understanding certain parts of this manual (and essential for Section 6) to read this paper. `QPE` uses the `ATMoSphere` inference engine as a substrate. Little knowledge of `ATMoSphere` is needed for routine `QPE` use, although a more detail understanding is essential to extending `QPE`.

By using explicit modeling and operating assumptions, the envisionment can be focused on particular subsets of possible behaviors. Furthermore, subsets of `QPE` can be used to answer simpler kinds of questions. For example, if only properties of single states are being explored, then limit analysis may be skipped for greater efficiency. Thus `QPE` can be used as a generator to see what qualitative states are consistent with assumed operating conditions of a system. (See Section 5 for details.)

`QPE`'s output can be used in many ways. A variety of routines are provided to simplify interactive probing of states and transitions, as well as the justifications which underlie them. A simple report generator for envisionments is provided. If you are using CLIM, the Common Lisp Interface Manager, there is a graphical interface for running `QPE` and some tools for plotting envisionments and other relevant information.

It should be noted that QPE is an implementation of one style of reasoning using QP theory. It is not identical to QP theory. We find envisioning extremely useful in developing new domain models and exploring modeling methodology, since envisioning exercises domain models more than other styles of reasoning. We also suspect that envisionments can be used as input for "compilers" that produce fast run-time programs for dealing with specific systems. However, it is important to realize that envisioning is not the only style of reasoning supported by QP theory (see [10] for details). Furthermore, while the modeling language provided in QPE is consistent with QP theory, there are many things one can say in QP theory that cannot be said in QPE's modeling language. The converse is also true: QPE's modeling language contains hooks for external representations and systems which are interpreted in specific ways that are not central to QP theory. (see [15] for details.)

1.2 Conventions

In addition to familarity with QP theory, we assume readers know Common Lisp (as defined by [21]). Computer output and data (i.e., scenarios and domain models) will be in fixed-width, typewriter style font, like this:

```
This could be computer output
     or part of a domain model  (But this is a comment!)
        or part of a scenario description.
```

Italics are used for comments, as shown above. For concreteness we use UNIX conventions when refering to pathnames.

Here is the minimum you need to know about ATMoSphere for routine operation of QPE and domain model construction. ATMoSphere is a descendent of AMORD [3], by way of RUP [19] and DEBACLE [9]. Like these systems, the model of data is individual assertions stored in a database which allows pattern-directed retrieval. Rules are run antecedently whenever data is found which matches their trigger conditions.

The differences in ATMoSphere are consequences of being built around an *assumption-based truth-maintenance system* [4]. Assertions take the form

(⟨*variable*⟩ . ⟨*value*⟩)

This may seem confusing at first, and this notion of variable should not be confused with the idea of a pattern variable. Here are some typical expressions printed as one might see them in a paper (or by the logic pretty-printer included in QPE) and their corresponding internal form:

Written form	Lisp form
Ds[Heat(F)]=1	((s (D (Heat F))) . 1)
Container(F)	((Container F) . :TRUE)
¢ontainer(F)	((Container F) . :FALSE)

While QPE takes pains to provide legible output, there is no parser to take external expressions and automatically produce the internal format. Some interface routines have limited "syntactic sugaring" for partially translating s-expressions to internal form. They work like this:

1. If the CAR of the expression is :==, assume that the CADR is a variable and the CADDR is its value.

2. If the CAR of the expression is NOT, assume that the CADR is a variable whose value is :FALSE.

3. Otherwise, assume that the expression is a variable whose value is :TRUE.

This data model reflects the internals of the ATMS. In particular, variables are associated with ATMS *classes*, and values are associated with ATMS *nodes*. You won't need to know much about this unless you start hacking QPE's internals. What you will need to know is that each assertion has an associated *class*, which is the leftmost non-list in the assertion's form.

A pattern variable in ATMoSphere is a symbol whose first letter is "?". In other words, ?x, ?A-variable, and ?1223 are variables, while X, x, and (foo bar) are not. There is no prohibition against including variables in database items. However, almost all ATMoSphere primitives treat database items as ground terms.

Rules in ATMoSphere have the following syntax:

(rule ⟨*condition*⟩ ⟨*triggers*⟩ . ⟨*body*⟩)

Essentially, ⟨*body*⟩ is executed for each environment formed by the variables in the patterns of ⟨*triggers*⟩ the first time the appropriate conditions hold. ⟨*body*⟩ can be arbitrary lisp code. ⟨*condition*⟩ defines when execution occurs. The following values are allowed:

:Intern Execute ⟨*body*⟩ as soon as the triggers are satisfied.

:In Execute ⟨*body*⟩ when the conjunction of the triggers is consistent.

:implied-by Execute ⟨*body*⟩ when every trigger is implied by a distinguished *focus environment.* See [16] for details.

QPE only uses :INTERN rules, the others are mentioned for completeness.

Each trigger takes the form

⟨*pattern*⟩ ⟨*options*⟩

where ⟨*pattern*⟩ is an s-expression which can contain variables, and ⟨*options*⟩ will be explained shortly. Bindings for triggers are found sequentially, in left to right order. The class of each pattern must either be a variable bound by a previous pattern or a constant; using a free variable for a class will result in an error. The (possibly empty) list of options consists of alternating keyword/value pairs. The following keywords are supported:

:var what follows is a variable which should be bound to the entire pattern.

:test what follows is an additional consistency test which is to be evaluated in the environment formed by all the triggers bound so far. If the test evals to NIL then the match is aborted, otherwise the match continues. *Warning: The expressions used in a* :test *should only operate on the structural form of the triggers. They should never test belief states or access the* ATMoSphere *database. Once aborted, the rule is never re-tested with that combination of triggers.*

There are three packages involved in QPE. The tms package contains the ATMS, the ADB package contains ATMoSphere and the qpe package contains qpe. qpe uses adb, so when in doubt try qpe as the package for the procedures below. Domain models are stored in the user package, wherever that is (generally user).

1.3 Legal Restrictions

The ATMS used in `ATMoSphere` is being made available courtesy Xerox PARC; please read and respect the copyright notice on it. In a better world (i.e., one less obsessed with litigation) the rest of this section would be unnecessary. However, in the United States one of our current cultural bugs makes the following copyright notice, which applies to both `QPE` and `ATMoSphere`, necessary:

Copyright ©1987 — 1992 by Kenneth D. Forbus

Permission to use this software is granted subject to the following restrictions and understandings:

1. This material is for educational and research purposes only, and should not be re-distributed in any form without prior written consent of Prof. Forbus.

2. Prof. Forbus has provided this software AS IS. Prof. Forbus has made no warrantee or representation that the operation of this software will be error-free, and he is under no obligation to provide any services, by way of maintenance, update, or otherwise.

3. Any user of such software agrees to indemnify and hold harmless Prof. Forbus and Northwestern University from all claims arising out of the use or misuse of this software, or arising out of any accident, injury, or damage whatsoever, and from all costs, counsel fees and liabilities incurred in or about any such claim, action, or proceeding brought thereon.

4. Users are requested, but not required, to inform Prof. Forbus of noteworthy uses of this software.

5. All materials and reports developed as a consequence of the use of this software shall duly acknowledge such use, in accordance with the usual standards of acknowledging credit in academic research.

6. In conjunction with products arising from the use of this material, there shall be no use of the name of Prof. Forbus, Northwestern University, nor of any adaptation thereof in any advertising, promotional, or sales literature without prior written consent from Prof. Forbus, or the University, in each case as appropriate.

1.4 Feedback

If you find bugs, have questions, or wish to make suggestions for changes or extensions to either QPE or this manual, please send electronic mail to:

`forbus@ils.nwu.edu`

Please note: When I was at University of Illinois, the address

`QPE@p.cs.uiuc.edu`

fed into a notesfile which was read by several people, including me. It turns out most people sent me bug reports via email directly anyway, so I haven't tried to set up the same mechanism here. If you do not have access to electronic mail, please use the following address:

Kenneth D. Forbus
Qualitative Reasoning Group
The Institute for the Learning Sciences Northwestern University
1890 Maple Avenue
Evanston, Illinois, 60201, USA

Fax: (708) 491-5258

Thank you for flying QPE.

2 Installing QPE at your site

To date, `QPE v2.6` has been successfully run on the following Common Lisp implementations:

- Symbolics Release 6 and Release 7
- Lucid 2.0 on SparcStations.
- Lucid 1.01 and 3.0 on the IBM RT
- Lucid 4.0 on the IBM RS/6000.

`QPE v2.6` was designed with portability in mind, so it should run on any Common Lisp implementation which rigorously follows [21]. (If you port it to a Common Lisp implementation not on this list I would appreciate hearing about it, especially if you have problems.)

2.1 Unwrapping the package

Generally you receive `QPE v2.6` as a UNIX tar file (or files). Untar'ing it will result in a number of directories being created which contain:

- Some utilities (default: `/u/utils/`)
- Johan's ATMS (default: `/u/atms/`)
- The ATMoSphere inference engine (default: `/u/adb/`)
- QPE itself (default: `/u/qpe/`)
- Domain files (default: `/u/qpe/domains/`)
- Example files (default: `/u/qpe/examples/`)
- Result files (default: `/u/qpe/runs/`)
- And some machine-specific graphical interfaces:
  - Symbolics release 6 (default: `/u/qpe/rel6/`)
  - Symbolics release 7 (default: `/u/qpe/rel7/`)
  - IBM RT running Lucid Common Lisp (default: `/u/qpe/rt`)

A complete listing of the files is in Appendix A.

To simplify life, we've used a system definition facility provided by Gregor Kiczales. It is a generalization of the facility he wrote for Portable Common Loops. Once you have decided where you want the files to to be, install those decisions according to the comments in the following files:

- `/u/atms/defsys.lisp`
- `/u/adb/defsys.lisp`
- `/u/qpe/defsys.lisp`

and if you are using machine-specific graphical interfaces, there will be either a `defsys` or a `defsystem` file in that directory that should also be edited.

2.2 Compiling and loading QPE

To load or compile QPE, you will need to first load `/u/utils/system.lisp` and then the three `defsys` files mentioned in the previous section.

Before doing anything else, make sure the variable `pcl::*pathname-extensions*` is bound appropriately. The `CAR` should be a string indicating the name for lisp source files on your system, the `CDR` should indicate the name for binary files on your system. Gregor supplied a table which contains conventions for many CL implementations, but there are some it doesn't know about. Compare the contents of `*features*` with the options in `system.lisp` to see what you might add to `*features*`.

Next, you must edit the defsys files in order to give them path information appropriate to your computing environment. For example, in the `defsys` file for QPE there are a number of variables which specify lists of files to use as domain models, scenario models, and so on. These must be updated according to where you put such information on your system. The easiest way to tell what to edit is to search for occurrences of `#+QRG-RT` in these files. Some of these indicate machine-specific dependencies (such as `+Symbolics` and `-Symbolics`, and those you should leave alone. But the others contain site information, and you should add information about your own site here. For example, if you are at University of North South Dakota at Whoople, you might change the specification of where QPE is stored as follows:

```
(defvar *qpe-pathname*
#+QRG-RT "/u/forbus/code/qpe/v2-6/"
#+KDF-RT "/u/qpe/v2-6/"
#+(and UIUC SYMBOLICS-RELEASE-7) "DOC:>QRG>qp>qpe>"
#+(and UIUC SYMBOLICS-RELEASE-6) "rube:>qp>qpe>v2-6>"
#+JOHAN "calvin:>qp>qpe>cl>"
#+PARC "bd:>qp>qpe>cl>"
#+UNSDW "/usr/local/qpe/"
)
```

Of course, you must push `:UNSDW` onto `*features*` before loading QPE, preferably in your lisp initialization file. *Warning: If you do not edit all of these pointers, you will almost certainly encounter very obscure bugs when trying to run* QPE*!*

Finally, you will need to compile the ATMS, then ATMoSphere, and then QPE. The following sequence of calls will do this:

```
(tms::recompile-atms)
(adb::recompile-adb)
(qpe::recompile-qpe)
```

If you have flamed out for some reason in one of these compilations and have made changes, you can re-start the compilation from an appropriate point by calling `(?XX::compile-?YY)` instead of `(?XX::recompile-?YY)`. Again, if the problem is due to a bug please let me know.

You may wish to compile domain files. To succeed, you must have QPE loaded and load the interpreted version of the domain file first. You may not have to load the domain file in future releases.

To load QPE all of the system definitions must be loaded. (If you have just compiled it this of course will already be the case.) Calling `(qpe::load-qpe)` will load QPE. You do not need to call the corresponding load procedures for the ATMS and `ATMoSphere`, they will be called automatically.

If you are compiling or loading a machine-specific graphical interface package, do so after loading QPE.

2.3 Installing machine-specific graphical systems

It should be clear how to load them from the sources. QPE can easily be run without a graphical interface, although graphics quickly becomes essential for inspecting envisionments. Currently there are graphical interfaces for:

- Symbolics Release 6.1, Release 7.2
- IBM RT using Lucid Common Lisp
- CLIM

The CLIM interface is now the prefered graphical user interface, and our future development will focus on this one. It has been tested under Lucid Common Lisp on the IBM RS/6000, but should be portable to CLIM implementations on other machines. There is a defsys file for the CLIM interface, and loading it should be self-explanatory.

2.4 Testing QPE

A simple benchmarking system is provided with QPE for testing and debugging. It is recommended that you run this system to check that the installation has not gone awry. The benchmarking system is described in more detail in Section 3.7; here we step through a simple session so you can test the envisioner.

The call `(qpe::benchmark-qpe)` begins an interaction which allows you to set up the particulars of a benchmarking run. These include comments, control of garbage collection, and so forth. When asked for a name for the run, give it some symbol which won't clash with files already on `/u/qpe/runs`[1]. Answer "yes" when asked whether you want to dump the results for each file. Answer "no" when asked whether you want to include all examples, and include examples one, two, and four.

When QPE is finished, check the summary file (called ⟨*yourfilename*⟩`.txt`) against the other `.txt` files on that directory. Your times may be different, of course. However, if the mode switches are the same, the corresponding entries in the columns marked `# sits`, `# sclasses`, `PC's`, `Qstate`, and `Ext`. `Qstate` should be the same. Otherwise, something is wrong.

[1] If your operating system places limits on the number of characters in a filename, it is helpful to keep the name short, since the files containing data for particular examples are named by appending "`-EXN`" to the file name, where `N` is the example number.

Figure 1: Scenarios are specified by asserting facts

Here is how a simple structural description of two containers sitting at the same height can be specified. This is a commented rendition of Example One of the benchmarks.

```
(assertq (state liquid))          ;; There is a state called "liquid"
(assertq (substance water))       ;; There is a substance called "water"
 ;; Now describe the individuals — F, G are containers, P1 a fluid path
(assertq (container F))
(assertq (container G))
(assertq (fluid-path P1))
 ;; P1 connects F to G and vice-versa
(assertq (fluid-connection P1 F G))
(assertq (fluid-connection P1 G F))
 ;pin down relative positions of containers and fluid paths
(assertq (equal-to (A (bottom-height f)) (A (max-height p1))))
(assertq (equal-to (A (bottom-height g)) (A (max-height p1))))
```

3 Operating QPE

This section focuses on how to run QPE. We discuss the format of scenario files first. Then we describe a simple character-oriented command interface that simplifies interacting with QPE when a graphical interface is not available. After describing routines for examining the envisionment and the underlying justifications, we list and explain the various modes QPE can operate in and what debugging flags are available. The report generator is described next, followed by a summary of the machine-specific graphical interfaces available. The benchmarking system, briefly described previously, is then explained in more detail. Finally, we end this section with some suggestions for operation and debugging gleaned from our experiences.

3.1 Inputs

QPE takes two inputs, a domain model and a scenario which will be understood in terms of that domain model. Domain models are the subject of Section 4, so we do not consider them further here. A scenario consists of a list of files, each describing some aspect of the particular situation being analyzed. These files can contain arbitrary lisp code. However, the following mixture of QPE and ATMoSphere primitives typically suffice:

`adb:assert!` *form* [*Procedure*]

`adb:assertq` *form* [*Macro*]

Declares that ⟨*form*⟩ holds universally. When used outside of rules ⟨*form*⟩ should be a ground term. `assert!` evaluates its argument, `assertq` evaluates only ATMoSphere variables. Figure 1 illustrates how a simple two-container problem can be represented this way.

`Set-Model-Assumption` *Assumption value* [*Macro*]

Modeling assumptions are explicitly represented in QPE. Each assumption can take on one of several value (propositional assumptions take on the values `:TRUE` or `:FALSE`). Normally QPE varies these background assumptions in the course of envisioning, so that every alternative is explored. This macro informs QPE that it should only explore the value you provide for that assumption.

3.2 The generic command interface

A simple character-oriented menu system is provided for day-to-day interactions with `QPE`. This system is machine-independent, and should run on any Common Lisp implementation. The generic menu commands are:

? Prints whatever help or documentation is available. (This command is available for all levels of the menu system, and is context-sensitive)

q Quits. This command is also available from all sub-menus.

An integer Selects the option corresponding to that integer. Typing **0** redisplays the options. Typing an integer past the range of available options causes re-prompting.

The entry into the `QPE` menu system is the following procedure:

`qpe` [*Procedure*]

This routine starts the character-oriented menu system. When reading characters, it is assumed that the current package is `QPE` (This may change in the future).

The commands listed in the menu are:

Reset Clears the current envisionment and initializes `ATMoSphere`.

ATMoSphere Clears caches for domain and scenario files.

Parameters Provides a menu for selecting domain and scenario files. The appropriate response to each option is a list of file names, entered by typing each file name on a seperate line. An empty line signals that the list is complete.

Modes Sets switches which effect `QPE`'s performance.

Flags Sets switches which print out additional tracing/debugging information.

Envision Performs the envisionment.

Show Shows various aspects of the envisionment and model.

Report Generates a report based on the current envisionment. Allows you to choose a report format, where the report should go, and add a line of comments. The contents of the report are explained in more detail below.

3.3 Interrogatives

These routines are designed to be called by modelers. Some of them are used in the report generator as well.

3.3.1 `ATMoSphere` interrogatives

Here we only mention `ATMoSphere` routines likely to be used in debugging domain models. There are two kinds of routines, those which examine the state of the ATMS and those which examine properties of the pattern-directed database. We begin with the ATMS oriented routines.

`e` ⟨*integer*⟩ [*Procedure*]

This procedure returns the environment whose unique index is ⟨*integer*⟩. For example,

`(e 3412)` $\longrightarrow$ `ENV-3412`

If no such environment exists it returns `NIL`. For convenience, this procedure exists in both the `ADB` and `QPE` packages.

`print-env` ⟨*environment*⟩ [*Procedure*]
Displays ⟨*environment*⟩, adding a suffix of "*" if it is contradictory.

`print-envs` ⟨*fact*⟩ [*Procedure*]
Displays the environments in ⟨*fact*⟩'s label. After each environment is printed it waits for a character before printing the next one. This feature can be turned off by supplying an optional second argument of NIL.

`contradiction-distribution` [*Procedure*]
Shows the number of contradictions broken down by number of assumptions. Handy for figuring out where to look for unwarrented contradictions.

`print-contradictions` ⟨*from*⟩ ⟨*to*⟩ [*Procedure*]
Shows the minimal contradictions from length ⟨*from*⟩ to ⟨*to*⟩.

The next set of interrogatives examine the state of the `ATMoSphere` database.

`show-classes` [*Procedure*]
This lists all `ATMoSphere` classes defined so far.

`show-class` ⟨*classname*⟩ *&key (forms nil) (rules nil)* [*Procedure*]
The class of a fact is the leftmost symbol. If facts of that class are defined, by default this routine shows how many instances and rules are associated with that class. Providing the optional keyword `:FORMS` with a non-nil argument displays all the instances. The keywork `:RULES` works similarly.

`print-status` ⟨*pattern*⟩ [*Macro*]
Shows all facts matching ⟨*pattern*⟩ and whether they are `IN` (i.e., have a non-empty label) or `OUT` (i.e., their label is empty).

`show-data` [*Procedure*]
Lists all facts, whether they are IN or not, and their labels. This procedure is only useful for simple experiments with `ATMoSphere`.

`adb::core-dump` ⟨*filename*⟩ [*Procedure*]
Dumps in printable form the contents of the `ATMoSphere` database. Useful for certain kinds of development and debugging situations. However, you really don't want to call this.

`whyq` ⟨*fact*⟩ [*Procedure*]
`why` ⟨*fact*⟩ [*Procedure*]
Shows the label and justifications for a fact. Sugars its input.

`justifications` ⟨*fact*⟩ [*Procedure*]
Lists the justifications for ⟨*fact*⟩ and the labels of their antecedents.

`consequences` ⟨*fact*⟩ [*Procedure*]
Lists the justifications which use ⟨*fact*⟩ in their antecedents.

3.3.2 Properties of the expanded domain model

QP theory allows the modeler to specify new abstractions which are to be instantiated on a scenario. These routines display information about these descriptions. Most of them can be accessed from the `QPE` interfaces.

`show-pis` *&optional (stream *standard-output*)* [*Procedure*]
`show-vis` *&optional (stream *standard-output*)* [*Procedure*]
`show-cdis` *&optional (stream *standard-output*)* [*Procedure*]
`show-pis` shows the process instances created by expanding the scenario with the domain model,

`show-vis` shows the view instances, and `show-cdis` shows both. (The union of views and process form the domain model's *conditionalized descriptions*, or "cd" 's.)

`show-choice-sets` *&optional (stream *standard-output*)* [*Procedure*]
Shows the sets of background assumptions created by model expansion.

`show-possible-directs` *&optional (stream *standard-output*)* [*Procedure*]
`show-possible-indirects` *&optional (stream *standard-output*)* [*Procedure*]
These routines show for each quantity what direct (or indirect) influences could potentially apply to it.

3.3.3 Global properties of the envisionment

`summarize-qpe-statistics` [*Procedure*]
Calls several other routines listed in this section to provide a synopsis of the current envisionment, as well as a summary of the underlying `ATMoSphere` statistics.
`print-QPE-modes` [*Procedure*]
Shows the current setting of the `QPE` mode flags.
`count-situations` [*Procedure*]
`count-sclasses` [*Procedure*]
Situations are particular qualitative states, sclasses are equivalence classes on qualitative states such that the same processes, views, and `Ds` values hold.. These routines return how many of them there are in the current envisionment.
`summarize-Ds-values` [*Procedure*]
Tabulates for each quantity the number of situations in which it takes on each possible value. The special value `BT` means the quantity does not exist in that situation.

3.3.4 Displaying information about quantities

`show-quantities` [*Procedure*]
Lists all quantities associated with the current scenario.
`show-number` ⟨*Anumber*⟩ [*Procedure*]
Shows the quantity space for that number. The entries `Above` and `Below` indicate the internal lexographic ordering between the numbers, *not* the order relationship that holds between them. If N_1 is above N_2 in the lexigraphic ordering, then the order relationship between them is indexed as N_1, N_2 rather than N_2, N_1.
`show-ineq` ⟨*pairofnumbers*, *inorder*⟩ [*Procedure*]
Shows the minimal environments which support each potential order relation.
`show-qstate` [*Procedure*]
Shows all "interesting" comparisons, broken down by their source.

3.3.5 Displaying information about situations and sclasses

`print-sclass` ⟨*sclass*⟩ [*Procedure*]
Prints information about an sclass, such as the individuals which exist in it, its duration, the number of situations it describes,and what view and process instances are active in it. A `status` of `PS-VS-COMPLETE` means that some `Ds` values are unknown, and other s-classes were introduced to represent different combinations of values for these unknowns. A `status` of `R-COMPLETE` means the `Ds` values were completely determined.

`sclass-r-completions` ⟨*sclass*⟩ [*Procedure*]
When ⟨*sclass*⟩ is `PS-VS-COMPLETE`, the result is the set of sclasses representing the consistent combinations of values for the unknowns. Otherwise, the result is `NIL`.
`print-ds-values` ⟨*sclass*⟩ [*Procedure*]
Shows the `Ds` values that hold in an sclass.
`env-sclass?` ⟨*environment*⟩ [*Procedure*]
`env-sclass` ⟨*environment*⟩ [*Procedure*]
Situations are represented by environments. `env-sclass` returns the situation associated with ⟨*environment*⟩. It produces an error if ⟨*environment*⟩ is not a situation, so if you are not sure use `env-sclass?` first, which will return `NIL` if it isn't.
`print-situation` ⟨*situation*⟩ [*Procedure*]
This produces the same information as `print-sclass`, except it also prints the details of ⟨*situation*⟩ as well.

3.3.6 Displaying information about limit analysis

`show-lhs` [*Procedure*]
Lists the limit hypotheses.
`princ-lh` ⟨*lh*⟩ *&optional (details? nil)* [*Procedure*]
Provides a one-line summary of ⟨*lh*⟩.
`print-lh` ⟨*lh*⟩ *&optional (details? nil)* [*Procedure*]
Provides a slightly longer printout about ⟨*lh*⟩. When `details?` is non-nil, the start and end environments are listed as well.
`print-situation-lh-table` [*Procedure*]
Shows what limit hypotheses potentially apply to each state.
`lh-stats` [*Procedure*]
Summarizes how many transitions were produced by the limit hypotheses, and how many transitions were pruned by each constraint.
`print-next-state-table` [*Procedure*]
Shows what transitions are possible from each state.

3.4 Modes

`QPE` includes two kinds of switches. *Flags*, described in the next section, print additional details about the internal operations of `QPE`. Changing the setting of a flag does not affect the answers produced nor any reports generated about the envisionment. *Mode switches* can change the way `QPE` runs in ways that can affect the results.[2] We group them according to what aspects of `QPE` they control.

3.4.1 Overall properties

`*Silence-zero-Ds-values*` [*Variable*]
Controls whether zero Ds values are printed. The corresponding menu item is `Skip printing zero Ds values`.
`*differential-situation-display*` [*Variable*]
Displays the environments constituting the situations for an sclass in terms of a base envirionment

[2] If you look through the source code you will see other internal switches not documented here. These internal switches shouldn't be fiddled with unless you really know what you are doing.

and difference environments. The base environment contains the common assumptions, with the difference environments highlighting the combination of assumptions unique to each situation. The corresponding menu item is `Display situations by differences`.

3.4.2 Generating View and Process structures

`*Assume-PCs*` [*Variable*]

When this switch is `NIL`, every statement in the preconditions of process and view instances is asserted to be universally true. This setting is only useful for debugging simple examples, and is mainly included for testing examples against `GIZMO`, our previous implementation of QP theory. The default is non-`NIL`. With this setting, preconditions are allowed to range over their possible values when building consistent collections of view and process structures. In the command menu this switch is called ``` ``Vary preconditions'' ```

3.4.3 Influence resolution

`*Resolve-Simple-Multiple-Directs*` [*Variable*]

When this flag is non-`NIL`, an assumption about relative magnitude is introduced whenever there is just one positive and just one negative direct influence on a quantity. If both rates have the same sign the newly introduced inequality assumption concerns their amounts, otherwise their magnitudes.

This switch is the sole survivor of a long process of experimentation. For instance, most early versions included a switch which searched for combinations of ordering information which would cancel for arbitrary combinations of direct influences, or even make assumptions about such orderings if necessary. Empirically, this was always a mistake. In `GIZMO` this effort was worthwhile, but in an ATMS there are too many consistent combinations. Every new assumption expands the solution space, since every situation must now include some choice for each new relationship. This switch constitutes a sensible compromise because one can easily imagine circumstances where a single piece of ordering information is available, but to get a large set of ordering information seems unlikely unless there is even more precise information available, and in that case extra work at a purely qualitative level seems less appropriate.

3.4.4 Limit analysis

`*Augment-Derivative-Comparisons*` [*Variable*]

Ensures that every relevant derivative relationship is known before limit analysis. This is important for two reasons. First, if the relationship between the derivatives for a pair of quantities is not known, no limit hypothesis can be established involving that pair, and hence potential transitions can be missed. Second, in some circumstances it turns out that no consistent relationship can hold between the derivative of some pair of quantities, hence that situation in fact is inconsistent and should be pruned. The default is that all relevant derivative relationships are found. However, in some cases this augmentation is irrelevant or too expensive, so it can be turned off by setting this switch to `NIL`.

`*Allow-Unstable-Trans*` [*Variable*]

When non-`NIL`, this flag allows transitions from unstable equilibriums. In particular, given quantities Q_1, Q_2whose amounts are compared, the following limit hypotheses are allowed:

$$A[Q_1] = A[Q_2] \qquad A[Q_1] > A[Q_2]$$
$$\longrightarrow$$
$$D[Q_1] = D[Q_2] \qquad D[Q_1] > D[Q_2]$$

$$\begin{array}{ccc} \texttt{A[Q}_1\texttt{]} = \texttt{A[Q}_2\texttt{]} & & \texttt{A[Q}_1\texttt{]} < \texttt{A[Q}_2\texttt{]} \\ & \longrightarrow & \\ \texttt{D[Q}_1\texttt{]} = \texttt{D[Q}_2\texttt{]} & & \texttt{D[Q}_1\texttt{]} < \texttt{D[Q}_2\texttt{]} \end{array}$$

The intuition is that such circumstances correspond to a metastable state. This flag can lead to weird behavior in circumstances involving changes in existence. On the menu this flag is called "`Allow Unstable Transitions`".

`*Case2-ECL-ON*` [*Variable*]

Case 2 of the Equality Change Law allows transitions back to equality in an instant when the difference between two numbers is "very small", i.e., they were influenced away from equality only for an instant. This law is somewhat controversial[3], so provision is made for turning it off. On the menu this switch is called ``` ``Equality Change Law, Case 2'' ```.

`*Allow-Coincidences*` [*Variable*]

When non-`NIL`, this flag allows simultaneous changes from and to equality. When `NIL`, limit hypotheses involving such changes are never combined into conjunctive changes. This has the effect of making changes from equality happen first. On the menu this switch is called `Allow Coincidences`.

`*Allow-discontinuous-directs*` [*Variable*]

When non-`NIL`, this flag does not prune transitions which violate continuity only with regard to directly influenced parameters. (Normally any transition between states which violates continuity at all are pruned.) On the menu this switch is called `Allow discontinuous direct influences`.

`*Prune-Asymptotic-Approaches*` [*Variable*]

When non-`NIL`, this flag prunes states which correspond to behaviors which approach, but never reach, a limit point. The detection criteria is that all limit hypotheses implied by the state are ruled out as inconsistent. On the menu this switch is called `Prune asymptotic approach states`.

This idea is a bit subtle. The intuition is that since limit analysis is a local computation it is fair to expect some transitions to be ruled out. But the existence of limit hypotheses means there are in fact limit points and they are being approached, and if none of them are reachable then the behavior is inconsistent. Consider a ball bouncing towards a wall. That behavior cannot continue unchanged forever — the ball must either hit the wall or change its behavior in some other way. This code essentially bars Zeno's paradox. Another example, due to Brian Falkenhainer, is a block attached to a string which is assumed to be completely inelastic. At the end of the drop the block's velocity must change instantaneously, so a transition to that state is ruled out via continuity. With this switch turned on, the state of the block falling but never falling so far that it reaches the length of the string is pruned from the envisionment.

3.4.5 The `QPE` Flight Recorder

p

There is always a trade-off between speed and safety. To simplify debugging it makes sense to perform extra (perhaps expensive) consistency tests and cache extra information about the internal operations of the program. To speed it up requires throwing out consistency tests that are unlikely

[3] QP theory assumed an infinitesimal model theory, but I never formalized it. Fortunately, others have [22,2], so that is not problematic. However, the phenomena of stutter, which led me to this choice in the first place, is. Brian Williams argued (personal communication) that I just wasn't enforcing continuity involving derivatives properly. Sure enough, when these constraints were tightened up the examples of stutter in [9] vanished! It can be proven that no stutter cycle of length two can exist, since it implies discontinuous derivatives. However, whether longer stutter cycles can exist is still an open question at this writing. If such a case ever arises, `QPE` will urge you to call me and tell me about it. Please do.

to help when you know what you are doing and caching as little as possible. QPE tries to provide the best of both worlds by allowing you to turn on *careful mode*. In careful mode many more consistency tests are performed, some of them quite computationally expensive. A *flight recorder* is also provided to record internal events in a database which can be inspected to determine what happened. A number of routines are provided for interacting with the database of events generated by the flight recorder.

`*Careful*` [*Variable*]

When non-NIL, the flight recorder is turned on. Events are added to the flight-recorder database and additional consistency tests are performed.

`types-of-events` [*Procedure*]

Lists the types of events in the flight recorder database.

`event-breakdown` [*Procedure*]

Shows how many events of each type are in the flight recorder database.

`show-events` *&optional (type nil) (filter nil)* [*Procedure*]

Prints events in order of occurrence. The `type` option restrict the events to be of a particular type. The `filter` option allows you to provide an arbitrary Lisp procedure to determine whether or not to include an event.

`list-events` *&optional (type nil) (filter nil)* [*Procedure*]

Like `show-events`, but produces a list as output.

`When-Careful` ⟨*body*⟩ [*Macro*]

Evaluates ⟨*body*⟩ only when careful mode is on. Good for adding your own consistency tests when extending QPE.

`add-recorder-event` *&rest event-description* [*Macro*]

Adds an event to the database. Each event must be a list, whose `CAR` is a symbol describing the type of events and whose `CDR` describes the event's particulars. Useful for extending QPE.

3.4.6 Report Generation

`*Silence-zero-Ds-values*` [*Variable*]

This switch shortens listings and screen displays by not showing `Ds` values which are zero. On the command menu this mode is labeled ``` ``Skip printing zero Ds values'' ```.

Two other switches will be added in the next version. These are

`*Dump-situations*` [*Variable*]

When NIL, the report only contains sclasses, not situations. This is useful when there are lots of situations whose differences aren't very interesting. Default is T.

`*Report-Mode*` [*Variable*]

Determines what formatting commands, if any, are used in the report file. Can be either `:ASCII` or `:LATEX`.

3.5 Flags

A number of flags are provided to control how much printing of QPE's internal state occurs during envisioning. These flags can be set by the appropriate command submenu, or directly with the procedure `adb::set-debug-flag`.

`adb::set-debug-flag` ⟨*flag*⟩ ⟨*value*⟩ [*Procedure*]

Sets debugging flag ⟨*flag*⟩ to ⟨*value*⟩. If ⟨*value*⟩ is NIL the flag is disabled, otherwise information associated with that flag will be printed. Typically this is called by some interface routine rather than directly by users.

If you are adding new code you might find the following information useful:

`adb::if-debug-flag` ⟨*flag*⟩ *&rest* ⟨*body*⟩ [*Macro*]
When ⟨*flag*⟩ is turned on, ⟨*body*⟩ is executed. It is assumed that the body only prints information to some stream, rather than doing signficant extra computation. (If you want to do something more intensive, please use the mode facility, especially `When-Careful`. See Section 3.4.5.)

`adb::*debug-table*` [*Variable*]
Internal use only. Stores setting of debug flags. Its value is either a list of symbols (the internal names for the flags) or unbound. A flag is considered to be set only if this list exists and the internal symbol corresponding to the flag is a member of that list.

3.5.1 Available flags

Here are the flags available from the menu system. The code contains other internal debugging flags which are not listed here, since they are rarely useful in figuring out modeling bugs.

Print notice of envisioning phases Shows what stage the envisionment is at.

Time QPE operations Times the major stages.

Quantity system Shows what inequalities are considered relevant.

Finding process and view instances Prints out terse description of a newly found instance.

Installing consequences of PI's and VI's States what process or view instance is being instantiated. Handy for narrowing down syntaxer bugs.

Details of PI and VI consequences This shows all justifications and nogoods created while instantiating view and process instances. Very handy for understanding how `QPE` works and for testing syntax extensions.

Break on finding process or view instance Most useful for figuring out why some instance is being built.

Break after scenario expansion. Allows you to poke around the newly-created descriptions without running the rest of `QPE`.

Sclass construction Marks progress in assuming preconditions, quantity conditions, and so forth.

Influence Resolution Prints possible `Ds` values found for each quantity.

Show justifications for no influences How the result is actually justified.

Show justifications for direct influences Similar.

Show justifications for indirect influences Similar.

Show justifications for ambiguous influence resolutions Shows search process as well as assumptions made.

Limit analysis Shows what limit hypotheses are found, outlines of operation.

Details of limit analysis Quite detailed, in fact.

Show results of Equality Change Law Shows how the Equality Change Law is applied during limit analysis.

Show all ATMoSphere nogoods (v. long!) Rapidly approaches the completeness and utility of a core dump. You shouldn't need this.

Show all ADB justifications (v. long!) Again, more information than one typically really wants to know. Useful mainly for understanding and debugging ATMoSphere or low-level QPE optimizations.

Incrementally monitor number of interpretations Shows number of solutions found in each round of interpretation construction.

3.6 Examining the output

There are several ways to use the envisionments produced by QPE. The interrogatives described in Section 3.3 provide the tools for on-line exploration. However, one also wants archival records of results, a way of generating hard-copy that can be perused on the train. The next section describes a user-extendable report generator which provides this service. Envisionments can be large and complicated, making graphical interfaces a necessity to figure out what is going on. We recommend the CLIM-based QPE GUI, which is described in a seperate document [20].

3.6.1 Report Generation

Reports can be generated either through the command menu or by calling the following procedure:
`dump-envisioner-results` ⟨*comments*⟩ ⟨*file*⟩ [*Procedure*]
Produces a report on the current envisionment which includes ⟨*comments*⟩ in the header page. The report is written to ⟨*file*⟩.

The report consists of several parts:

Header page Describes when and where the envisionment was generated and summarizes its statistics.

ATMoSphere statistics See [17] for an explanation.

Instances listing Shows what view instances and process instances were created. The short forms (i.e., PIO) are used in the rest of the listing to save page space.

Quantity space listing Shows all relevant inequalities, and what part of the computation deemed them relevant.

Limit hypothesis concordance Defines the short names used in the rest of the listing. More details concerning the hypotheses are included later.

Transition table Lists the transitions comprising the envisionment, indexed by situations.

Situation listing Describes each sclass in detail. The environments listed with each sclass are the situations described by that sclass.

Limit hypothesis listing Describes each limit hypothesis in detail, including the consistent starting and end environment.

Direct influencers table Lists the potential direct influences on each quantity, if any.

Indirect influencers table Lists the potential indirect influences on each quantity, if any.

One internal flag is included to change the basic report format:

`*dump-situations*` [*Variable*]

When non-`NIL`, include situation envirionments in the dumped description of each sclass.

For each section there are enough titles and dividers that character-oriented programs should be able to perform simple analyses (such as source compares). Of course, any fixed report format is likely to leave something out. A facility for adding your own information into the report is provided to get around this in simple cases.

`*user-report-hooks*` [*Variable*]

Either `NIL` or a list of procedures which define addendums to a report. Each procedure must take only one argument, the output stream for its results. Page breaks will be inserted before each addendum.

`defUser-Report-Generator` ⟨*procedure*⟩ [*Macro*]

Adds ⟨*procedure*⟩ to the list of addendum generators.

3.7 The benchmarking system

The benchmarking system provides a means of running several examples in sequence without intervention. It is very handy when testing changes in a model (i.e., you can run a sequence of test examples and compare the results to see what is different) or when trying to extend or modify `QPE`.

`benchmark-qpe` [*Procedure*]

Asks a sequence of questions about the common parameters of the run, including how much information should be dumped, control of garbage collection (depending on implementation), and so forth. Then it asks which of a series of examples you wish to run. Then it proceeds to run them. It is assumed that any debugging flags or mode switches were properly set before calling `benchmark-qpe`.

`rerun-qpe` *&optional* ⟨*ResultFile*⟩ [*Procedure*]

If you want to run `QPE` again with the same benchmark parameters (for example, you wanted to change debugging flags or mode switches), just call this procedure. If you are systematically exploring some variation of parameters or switch settings you may want to store results of different runs in different files. That is what the optional argument ⟨*result − file*⟩ is for.

`*benchmark-scenarios*` [*Variable*]

Contains a list of scenarios in the following format:

(⟨*prettyname*⟩ ⟨*abreviation*⟩ . ⟨*scenariofiles*⟩)

where ⟨*prettyname*⟩ is what is displayed when interacting with `benchmark-qpe`, ⟨*abreviation*⟩ is a suffix added to the name of the run to distinguish data files for that example from other examples, and ⟨*scenariofiles*⟩ is a list of files comprising the scenario. The default value for this variable is a set of examples that we use for debugging `QPE`.

3.8 The Event Scheduler

When using QPE as a module, or when experimenting with augments,[4] it is often handy to be able to perform operations just before or just after stages of the envisioner's operation. For instance, if one were searching a space of modeling assumptions to find an envisionment that is rich enough to

[4] See `syntaxer.lisp` for a clean way to add your own code to `QPE`. Augments defined with this facility will show up in reports, for example.

make an accurate prediction yet not drown in details, you might want to check the states generated after influence resolution to see whether or not it is worth going on to limit analysis. This code provides a place to hang such hooks. It is used internally by several facilities, such as the closed-world set assumption mechanism.

`schedule-qpe-event` ⟨*thunk*⟩ ⟨*modifier*⟩ ⟨*operation*⟩ ⟨*priority*⟩ [*Procedure*]
where ⟨*thunk*⟩ is a procedure of no arguments, ⟨*modifier*⟩ is `:BEFORE` or `:AFTER`, ⟨*operation*⟩ is a named QPE operation (like `:LIMIT-ANALYSIS`), and ⟨*priority*⟩ is an integer. This will cause ⟨*thunk*⟩ to be executed either before or after the named operation. ⟨*priority*⟩ specifies the order in which thunks are executed, with smaller numbers being executed before larger ones.

`show-qpe-internal-events` ⟨*&optional*⟩ ⟨*stream*⟩ [*Procedure*]
Prints the contents of the table for debugging purposes.

4 Building Domain models

The methodology of qualitative modeling is still in its infancy. Nevertheless, several views have been proposed and are being explored. We begin by summarizing some of the modeling philosophy in QP theory, and then illustrate how `QPE` supports that philosophy. QP theory does not completely specify a modeling language; many constraints arise from assumptions about the kinds of problems to be solved or the available implementation. The particular modeling language built in to `QPE` is described next, including a complete listing of built-in syntactic constructs for definping domain models. Since `QPE` is designed as an experimental tool, we also describe how users may extend the modeling language by adding new syntactic constructs. We also provide some hints on building domain models, based on our experience in several projects.

4.1 An overview of the modeling process

We call the system or situation being modeled the *scenario*, and its qualitative model the *scenario model*. The simplest way to build a scenario model is to create a model of that specific scenario for a particular purpose. While useful systems may be built this way, it is also easy to generate ad hoc models of dubious merit, where the model must be thrown away whenever the scenario or purpose changes slightly. An indirect route is more robust – build first a general-purpose *domain model*, which describes a class of related phenomena or systems. Ideally, a scenario model can be built by instantiating and composing descriptions from the domain model. Developing a domain model requires more initial work, but it simplifies generating models for a range of scenarios. Furthermore, ad hoc aspects of models are more likely to be discovered if the same descriptions are re-used in a variety of settings.

QP theory is designed to support the creation of problem-independent domain models that include explicit simplifying assumptions and modeling assumptions (e.g., [7]. This accounts for much of the complexity of `QPE`.

4.2 Built-in primitives for domain model specification

Examples of how these primtives are used can be found in the sample domain files distributed with `QPE`. When in doubt, check the source code in the file `syntax.lisp`.

4.2.1 Declarations

`defQuantity-Type` ⟨*name*⟩ *&rest* ⟨*arguments*⟩ [*Macro*]

Declares ⟨*name*⟩ to be a (logical) procedure whose range is the set of quantities. The domain is specified by ⟨*arguments*⟩, which declares each argument as either `individual` or `constant`. The distinction is important because a quantity can only exist when all of the individuals it depends upon do.

`DefComputable-Relation` ⟨*predicate*⟩ [*Macro*]

Declares that all assertions whose class is ⟨*predicate*⟩ will be known to either be true or false in any situation. What this means is that `QPE` will not introduce assumptions for facts of this form, even if they appear in preconditions. This is a useful efficiency trick. However, you will almost certainly get an error if there is some legal situation for which the fact is not known.

`defProcedure-name` *thing* [*Macro*]

This is used by the syntaxer in desugaring expressions. If a functor is not known to be a procedure name, it is assumed to be a predicate.

`defModelingAssumptionType` ⟨*rel*⟩ [*Macro*]
States that all assertions whose functor is ⟨*rel*⟩ should be added to the choice sets which form the constituents of state.
`defModelingAssumption` ⟨*form*⟩ [*Macro*]
States that ⟨*form*⟩ is a proposition which should be added to the choice sets which form the constituents of state.
`defentity` ⟨*predicate*⟩ *&rest* ⟨*consequences*⟩ [*Macro*]
If ⟨*predicate*⟩ is a symbol, then ⟨*predicate*⟩ is a one-place predicate on individuals such that (a) when it holds the consequences believed and (b) when it holds the individual refered to must exist. The canonical variable `?self` is used inside ⟨*consequences*⟩ to refer to the generic individual in this form. If ⟨*predicate*⟩ is a list, then the `CAR` of that list is treated as the predicate and the `CADR` becomes a form which specifies the individual. This version is useful for defining compound objects (like contained stuffs; see `nst.lisp` for examples).
`defpredicate` *predicate &rest consequences* [*Macro*]
Pretty much the same as `defentity`, but with no presumption that its argument is an individual or if it is that there is any relation between its existence and this predication. (This means you had better enforce it yourself or some fairly counter-intuitive things can occur)

4.2.2 Defining domain-specific choices

Many of the choice sets which comprise the solution space are built into `QPE`, such as the relationship between limit points. `QPE` also allows user-declared choice sets as a means of interfacing with external representations.
`Choice-set` ⟨*term*⟩ [*Syntax*]
Declares that the potential values for ⟨*term*⟩ form a choice set. All assertions of the form

$$\langle term \rangle = \langle value \rangle$$

will be treated as mutually exclusive assumptions, and every situation will include (or imply) exactly one of the assumptions from this set.
`Choices` ⟨*term*⟩ ⟨*alternatives*⟩ [*Syntax*]
Like `Choice-set`, but ⟨*alternatives*⟩ explicitly lists the possible values.
`Switch` ⟨*term*⟩ [*Syntax*]
Much like `Choice-set`. Provides a hook for generating action-augmented envisionments (see [14]). The intuition is that a switch is something that can be "flipped" by an agent interacting with the world. `STRIPS`-like operators are provided that flip switches as a way of integrating actions into qualitative simulation.
`Settings` ⟨*SwitchTerm*⟩ ⟨*SettingsList*⟩ [*Syntax*]
States that the switch ⟨*SwitchTerm*⟩ can have the settings ⟨*SettingsList*⟩.

4.2.3 `ATMoSphere` connections

Some of these are syntactic sugar. More such sugar will be added in the future, since I think it is important to make implemenation-independent domain models. But it takes time and experience to develop the right conventions, so right now this is minimal.
`:==` ⟨*variable*⟩ ⟨*value*⟩ [*Syntax*]
Syntactic sugar for

`(` ⟨*variable*⟩ `.` ⟨*value*⟩`)`

`Rule` ⟨*triggers*⟩ *&rest* ⟨*body*⟩ [*Syntax*]
Equivalent to `adb::Rule`. The forms in ⟨*body*⟩ are *not* currently translated by the syntaxer. This may change in the future.

4.2.4 Connectives

Because it used an LTMS, `GIZMO` allowed a subset of predicate calculus, including restricted quantifiers, in domain models. The use of horn clauses in the ATMS means that the implementation of logical connectives is restrictive at best. Experiments are underway to efficiently fix this. As warrented, the interpretation of connectives will be broadened and old connectives turned back on.

`And` ⟨*conjuncts*⟩ [*Syntax*]
Usual meaning. Rarely needed explicitly in domain models, since the contents of the fields of QP descriptions are implicitly conjunctive.

`When` ⟨*antecedent*⟩ *&rest* ⟨*consequences*⟩ [*Syntax*]
Ensures that ⟨*consequences*⟩ are believed when (a) the `When` itself is believed and (b) ⟨*antecedent*⟩ is believed.

`Or` ⟨*disjuncts*⟩ [*Syntax*]
Not available yet. An error will be signaled if you use this.

`Taxonomy` ⟨*alternatives*⟩ [*Syntax*]
Not available yet. An error will be signaled if you use this.

`At-Most-One` ⟨*alternatives*⟩ [*Syntax*]
Not available yet. An error will be signaled if you use this.

`Not` ⟨*proposition*⟩ [*Syntax*]
Flips the truth value associated with ⟨*proposition*⟩, i.e., `:TRUE` becomes `:FALSE` and vice-versa.

`->` ⟨*antecedent*⟩ ⟨*consequent*⟩ [*Syntax*]
Not available yet. An error will be signaled if you use this.

`iff` ⟨*prop*1⟩ ⟨*prop*2⟩ [*Syntax*]
Not available yet. An error will be signaled if you use this.

4.2.5 Basic QP primitives

Quantities and Numbers Many of these constructs are described in more detail in Section 2 of [10].

`defQuantityType` ⟨*qname*⟩ *&rest* ⟨*ArgTypes*⟩ [*Syntax*]
Declares that ⟨*qname*⟩ is a procedure whose range is quantities. ⟨*ArgTypes*⟩ declares the procedure's domain by specifying whether or not each argument is an `individual` or `constant`. This distinction is important for enforcing the relationship between continuous properties and existence. (See Section 7, [9].)

`Quantity` ⟨*quantity*⟩ [*Syntax*]
One-place predicate used to introduce continuous properties of individuals.

`A` ⟨*quantity*⟩ [*Syntax*]
The amount of a quantity. Amounts are numbers.

`D` ⟨*quantity*⟩ [*Syntax*]
The derivative of a quantity. Derivatives are numbers.

`s` ⟨*number*⟩ [*Syntax*]
The sign of a number.

`m` ⟨*number*⟩ [*Syntax*]
The magnitude of a number.

Relationships between numbers These relationships form a qualitative language for describing numbers and differential equations.

`less-than` $\langle N_1 \rangle$ $\langle N_2 \rangle$ [*Syntax*]
States that $\langle N_1 \rangle < \langle N_2 \rangle$

`equal-to` $\langle N_1 \rangle$ $\langle N_2 \rangle$ [*Syntax*]
States that $\langle N_1 \rangle = \langle N_2 \rangle$

`greater-than` $\langle N_1 \rangle$ $\langle N_2 \rangle$ [*Syntax*]
States that $\langle N_1 \rangle > \langle N_2 \rangle$

`Qprop` $\langle$*constrainee*$\rangle$ $\langle$*constrainer*$\rangle$ [*Syntax*]
Equivalent to

$$\langle \textit{constrainee} \rangle \propto_{Q+} \langle \textit{constrainer} \rangle$$

`Qprop-` $\langle$*constrainee*$\rangle$ $\langle$*constrainer*$\rangle$ [*Syntax*]
Equivalent to

$$\langle \textit{constrainee} \rangle \propto_{Q-} \langle \textit{constrainer} \rangle$$

`Procedure-Spec` $\langle$*name*$\rangle$ *&rest* $\langle$*constraints*$\rangle$ [*Syntax*]
$\langle$*name*$\rangle$ is a procedure specified by $\langle$*constraints*$\rangle$, at least one of which is either a `Qprop` or `Qprop-`. Currently $\langle$*constraints*$\rangle$ can only contain qualitative proportionalities. See Section 2.6 of [10] for details.

`Correspondence` *&rest* $\langle$*pairs*$\rangle$ [*Syntax*]
Pins down a particular point on a monotonic procedure defined by qualitative proportionalities. See Section 2.6 of [10] for details.

`Q=` $\langle Q_1 \rangle$ $\langle Q_2 \rangle$ [*Syntax*]
$\langle Q_1 \rangle$ must be a quantity. $\langle Q_2 \rangle$ is either a quantity or an *arithmetic form*. When $\langle Q_2 \rangle$ is a quantity, it means:

`A[` $\langle Q_1 \rangle$ `]` = `A[` $\langle Q_2 \rangle$ `]`
`D[` $\langle Q_1 \rangle$ `]` = `D[` $\langle Q_2 \rangle$ `]`

An arithmetic form is the sum or difference of two quantities. When $\langle Q_2 \rangle$ is an arithmatic form, it means that $\langle Q_1 \rangle$ is equal to the sum or difference specified by $\langle Q_2 \rangle$. Sums and differences are not implemented as distinct constraints in basic QP theory, but are expanded by the syntaxer into qualitative proportionalities and correspondences. (No qualitative proportionality is installed when $\langle Q_2 \rangle$ is a quantity.)

`=` $\langle N_1 \rangle$ $\langle N_2 \rangle$ [*Syntax*]
Rarely useful for modeling. If $\langle N_2 \rangle$ is not a relational form, this is equivalent to using `equal-to`.

`I+` $\langle influenced \rangle$ $\langle influencer \rangle$ [*Syntax*]
See Section 3.2 [10] for details.

`I-` $\langle influenced \rangle$ $\langle influencer \rangle$ [*Syntax*]
See Section 3.2 [10] for details.

`+` $\langle NorQ_1 \rangle$ $\langle NorQ_2 \rangle$ [*Syntax*]
Usual interpretation. Expands internally to `++rel`, to prevent duplicate expansion.

`-` $\langle NorQ_1 \rangle$ $\langle NorQ_2 \rangle$ [*Syntax*]
Usual interpretation. Expands to `--rel`, to prevent duplicate expansion.

`++rel` [*Syntax*]

`--rel` [*Syntax*]

`+rel` [*Syntax*]

`-rel` [*Syntax*]
Internal forms for expanding `+` and `-`.

`DefAlgebraic-Operator` *symb* [*Macro*]
(Say something about what this is.) The current list of algebraic operators is `+`, `-`, `=`, `Sum-of`, `Qprop`, and `Qprop-`

4.2.6 Existence

Physical objects can have finite temporal extent. Capturing this aspect of the world requires reasoning about existence.

`Exists` ⟨*obj*⟩ [*Syntax*]
Indicates that ⟨*obj*⟩ physically exists when true.

`Introduces` ⟨*obj*⟩ [*Syntax*]
When true, `Introduces` declares ⟨*obj*⟩ to be an individual which physically exists. Can only be used within a view or process.

`There-Is` ⟨*obj*⟩ [*Syntax*]
Synonym for `Introduces`.

`Uniquely-Introduces` ⟨*obj*⟩ [*Syntax*]
This predicate can only be used within a process or view. It indicates that ⟨*obj*⟩ physically exists if and only if that process or view is active.

`There-Is-Unique` [*Syntax*]
Synonym for `Uniquely-Introduces`.

`Only-During` ⟨*fact*⟩ [*Syntax*]
When used in a view, process, or other `QPE` defining form, indicates that ⟨*fact*⟩ holds if and only iff the defining form holds.

4.2.7 Defining domain-specific sets

`Set` ⟨*SetName*⟩ *&optional* ⟨*mod*⟩ ⟨*op*⟩ ⟨*pri*⟩ [*Syntax*]
Defines ⟨*SetName*⟩ to be a set. Members of the set are specified by the form

`(` ⟨*SetName*⟩ `HAS-MEMBER` ⟨*member*⟩`)`

and a construal of a set (i.e., the items held to be the complete list of members of a set after making closed-world assumption) is specified by the form

`(` ⟨*SetName*⟩ `MEMBERS` ⟨*TheMembers*⟩`)`

The optional arguments ⟨*mod*⟩, ⟨*op*⟩, and ⟨*pri*⟩ specify when the set should be closed, using the conventions of the `QPE` event scheduler. The default is after scenario expansion, with a priority of 100.

4.3 Adding new primitives

`DefTranslation` *Need argument list here* [*Macro*]
TO BE DESCRIBED

`debug-QPE-syntaxer` [*Procedure*]
TO BE DESCRIBED

`DefRelational-Operator` *symb inf* [*Macro*]
TO BE DESCRIBED

5 Tips, tricks, and traps in using QPE

This section helps you get the most done with QPE by summarizing some of the common mistakes and problems people have when first using the system, and some help on how to debug and work around such problems.

5.1 Getting off the ground

Fortunately, most Common Lisp implementations are of sufficiently high quality that QPE typically operates "right out of the box" these days. However, if you have just set up QPE for the first time, or ported it to a new machine, please try the benchmark examples as described in Section 2.4. If the data you get does not match the examples in the runs directory (i.e., number of situations and sclasses, sizes of preconditions, quantity spaces, and other comparisons), then there is probably a problem to resolve with the port before trying your own examples.

Portability in any computing environment seems to be approached asymptotically; some environments which still cause trouble include:

Symbolics CLOE environment on 386's: de Kleer's ATMS has many Symbolics-specific optimizations which assume Symbolics hardware, and are keyed to the `:SYMBOLICS` tag on the features list. We have run QPE on an IBM PS/2 in this environment, using DeCoste and Collins' CATMS instead of de Kleer's ATMS, however.

Mac Common Lisp: Consumer code in ATMS v22 blows out the compiler. Might be fixable, see de Kleer.

Some KCL implementations: Results with KCL have been mixed.

If you think you are having trouble with ATMoSphere, please use `adb::shakedown`, which runs tests which stress various parts of ATMoSphere to help pinpoint the trouble. The ATMoSphere file `debug.lisp` has some other debugging procedures which may prove useful.

5.2 Model building problems

Problem: A model fragment (view or process) isn't being instantiated when it should be.

Domain theories and scenario models should be in the user package. This means that predicates like `correspondence` had better be `user::correspondence` as opposed to `hal::correspondence`, for instance. The standard QPE modeling predicates are imported into the `qpe` package from the `user` package, so they may be used freely in code which extends QPE.

If you think this might be your problem, call `qpe::check-for-overlapping-classes`, which finds any ATMoSphere classes which have the same name but are in different packages.

Problem: QPE *seems fussy about the order of terms in an inequality statement.*

Ordinal statements are canonicalized for efficiency. That is, if N_1 is less than N_2 in the lexigraphic order defined by the Common Lisp ordering on strings, QPE will be happy with the statement `(Greater-Than` N_1 N_2`)` but not `(Less-Than` N_2 N_1`)`.

Problem: The ATMS errors out, claiming a total contradiction.

The distinction between assumptions and premises is important: This error indicates that the empty environment has been found to be inconsistent, which means that no larger environment can be consistent, so there is basically nothing the ATMS can do. Introducing too many premises, as opposed to assumptions, is always the culprit here. Premises are made using `assert!` (or `rassert!`). Such statements become universally held beliefs, that is, they hold in the empty environment (`ENV-2`). Assumptions are made using `assume!` (or `rassume!`). Assumptions are the stuff of which environments are made; contradictory assumptions are noted by the ATMS in its internal nogood database, but this is normal and desirable.

The only premises in your domain theory should be declarations of constants (e.g., `(Phase liquid)`). The only premises in your scenario models should be structural descriptions (e.g., `(container boiler)`), modeling assumptions (e.g., `(Consider (Faults Compressor))`), or operating assumptions (e.g., `(Less-Than (A (pressure atmosphere)) (A (pressure (Outlet Turbine))))`). Typically the contradiction arises because of the indirect consequences of these premises. One way to figure out where this is happening is by stepping through the implications of your premises using `adb::consequences` (or a ZGRAPH dependency plot). Another way to localize the problem is to change premises into assumptions and have `QPE` break after scenario expansion. The nogood database at that stage should contain the combinations of statements which are contradictory.

Problem: Domain theory or scenario model is getting too complex to manipulate.

Use multiple files for domain theories and examples: This is a standard programming technique which is often overlooked when building models. Splitting a domain theory into pieces allows alternate versions to be explored more easily. Splitting a scenario model into distinct files allows extra constraints or simplifying assumptions to be added as necessary.

A natural way to carve up a domain model is by definitions of object types and processes. For instance, a thermodynamics model might define sources and sinks in a seperate file from the definitions of fluids, and group the physical processes into files by some organizing principles (i.e., phase changes versus flows). In our experience, a piece of a domain theory which is over three pages long is a good candidate for decomposing into more than one file. A natural way to decompose an example is to use one file for the structural description and other files for different simplifying assumptions.

5.3 When the envisionment gets too large

Envisioning is an exponential process, and so must be invoked with care. Fortunately, many real tasks impose simplifying assumptions which make envisioning tractable. For example, quantitative analyses of thermodynamic cycles focus on steady-state behaviors. `QPE` can easily be constrained to simply generate steady-state behaviors, and even quite complicated scenarios run quite fast under such assumptions. (The file `qrules.lisp` implements the built-in vocabulary for making steady-state assumptions.)

Use lowest degree of resolution consistent with getting useful results. If you only want to know that a state consistent with a conjunction of facts exists, then you may not care about every `Ds` value, and hence could improve performance by just branching on ambiguous influences.

Explore large envisionments sequentially. By using extra files to introduce simplifying assumptions you can precisely control which subspace of the envisionment is generated. This means you can explore an envisionment that would otherwise be too large by exploring it one piece at a

time. (The major problem with this approach is that you will not have the transitions between the subspaces.)

Figure out where your model is underconstrained. Use the interface procedures which display the list and sizes of choice sets to figure out which parts of your model might be constrained by additional operating assumptions. (The procedure `qpe::check-psvs-size` provides this service.)

5.4 When the envisionment isn't large enough

Sometimes the envisionment fails to contain states or transitions that the modeler's intuitions says should be there. There are several ways to track these problems down:

Explore the nogood database to find out why states are inconsistent. There are several ways to do this. The `envs` command on the CLIM GUI can be used to build environments which contain facts that partially define a state. If the environment so built is inconsistent, the procedure `adb::track-down-killer` can find the smallest subset which is inconsistent. A list of nogoods which include all the facts in a list of statements can be found by calling `adb::facts-nogoods`. In the worst case, `adb::print-contradictions` can be used.

Use the `QPE` flight recorder. All states and sclasses and transitions that are pruned from the envisionment are recorded along with the reason for their invalidation if `QPE` is being run in careful mode.

Use special-purpose debugging procedures: The procedure `qpe::check-transitions-for` uses `QPE` internal datastructures to reconstruct why invalid transitions from a given situation were ruled out.

6 Using QPE as a module

If you are using QPE to build and refine domain models then you probably don't need to know this. If you are trying to extend QPE or using it as a module in a larger system then you probably do.

6.1 Internal representations

The details of these internal datastructures is likely to change from version to version.

`ineq` [*Structure*]
Internal representation of the ordering relationships that can hold between a pair of numbers.

`QP-inequality?` [*Procedure*]
Returns non-NIL if the argument is an `ineq` struct.

`quant` [*Structure*]
Internal representation of a quantity.

`QP-quantity?` [*Procedure*]
Returns non-NIL if the argument is an `quantity` struct.

`qnum` [*Structure*]
Internal representation of a number.

`QP-number?` [*Procedure*]
Returns non-NIL if the argument is an `number` struct.

`sclass` [*Structure*]
Internal representation of an sclass.

`sclass?` [*Procedure*]
Returns non-NIL if the argument is an `sclass` struct.

`lh` [*Structure*]
Internal representation of a limit hypothesis.

`limit-hypothesis?` [*Procedure*]
Returns non-NIL if the argument is an `lh` struct.

`*zero-struct*` [*Variable*]
Points to the special QP number `ZERO`.

`*next-state-table*` [*Variable*]
Alist whose key is a situation and whose item is an index of what transitions are deemed possible from that situation. The index is another alist whose key is a limit hypothesis and whose item is a situation or situations.

`*named-choice-sets*` [*Variable*]
Lists the kinds of choice sets of the domain.

`*choice-sets*` [*Variable*]
Lists the choice sets for the scenario model.

`*lisp-package*` [*Variable*]
Bound to the lisp package for the particular implementation.

`*user-package*` [*Variable*]
Bound to the user package for the particular implementation.

`*qpe-pathname*` [*Variable*]
Bound to the QPE package for the particular implementation.

`*domain-path*` [*Variable*]
String indicating the directory where domain model files are found.

`*examples-path*` [*Variable*]
String indicating the directory where example files are found.

`*QPE-rules-files*` [*Variable*]
List of file names for rules used internally by `QPE`.
`*domain-files*` [*Variable*]
List of files comprising the current domain model.
`*scenario-files*` [*Variable*]
List of files comprising the current example.
`*benchmark-scenarios*` [*Variable*]
Alist of examples used by the benchmarking system.
`*benchmark-physics-files*` [*Variable*]
List of Domain model files to be used for benchmarking.
`*QPE-benchmark-data-directory*` [*Variable*]
String indicating the directory for benchmark results.

6.2 Interface procedures

Here are the procedures and variables likely to be useful for someone incorporating `QPE` as a module in their own system. More documentation will be provided in a future version of this manual.

`DefQPEmode` [*Macro*]
`put-sclass-prop` *(sclass pval pname)* [*Procedure*]
`get-sclass-prop` *(sclass pname)* [*Procedure*]
`get-lh-prop` *(lh pname)* [*Procedure*]
`put-lh-prop` *(lh pval pname)* [*Procedure*]
`list-all-sclasses` [*Procedure*]
`get-individuals` [*Procedure*]
`get-pis` [*Procedure*]
`get-vis` [*Procedure*]
`gather-preconditions` [*Procedure*]
`lookup-sclass` *(keys)* [*Procedure*]
`map-sclasses` *fun &optional (sits *ps-vs*)* [*Procedure*]
`map-situations` *fun* [*Procedure*]
`put-ineq-prop` *ineq val prop* [*Procedure*]
`get-ineq-prop` *ineq prop* [*Procedure*]
`gather-quantities` [*Procedure*]
`reload-QPE-rules` [*Procedure*]
`*QPE-initialized*` [*Variable*]
`*Domain-loaded*` [*Variable*]
`*Scenario-loaded*` [*Variable*]
`*Analysis-complete*` [*Variable*]
`start-qpe` [*Procedure*]
`load-domain` [*Procedure*]
`load-scenario` [*Procedure*]
`expand-scenario` [*Procedure*]
`envision` [*Procedure*]
`limit-analysis` [*Procedure*]
`quantity-type?` *x* [*Procedure*]
`quantity-of` *form* [*Procedure*]
`make-derivative` *Qty* [*Procedure*]
`make-amount` *qty* [*Procedure*]

`derivative?` *qref* [*Procedure*]
`amount?` *qref* [*Procedure*]
`special-quantity?` *q* [*Procedure*]
`view-qnum` *qnum* [*Procedure*]
`put-qnum-prop` *qnum val prop* [*Procedure*]
`get-qnum-prop` *qnum prop* [*Procedure*]
`ineq-num1` *ineq* [*Procedure*]
`ineq-num2` *ineq* [*Procedure*]
`*PS-VS*` [*Variable*]
This is a list of distinct S-complete Sclasses.
`*PS-VS-Envs*` [*Variable*]
`*BASIS*` [*Variable*]
Background assumptions for the envisionment. Default value is `tms::*empty-env*`
`*ENFORCE-ENV*` [*Variable*]
Environment representing the assumption that the laws governing the interaction of being a quantity and existence of the object a quantity pertains to are in force. The negation of this assumption is sometimes used when checking continuity for objects that cease to exist.
`*ENVISION-ENV*` [*Variable*]
;; final environment
`*domain-expander-forms*` [*Variable*]
;; User-hooks
`*quantity-types*` [*Variable*]
;list of declared quantity types
`*explicit-procedures*` [*Variable*]
;list of declared explicit procedures
`DefAugment-File` *file &key (init nil)* [*Macro*]
`*Augment-Files*` [*Variable*]
"Files containing user-supplied code."
`DeclareAugment-File` *file* [*Macro*]
`reload-augment-files` [*Procedure*]

7 Plans for future releases

I plan to produce perhaps one more version of QPE, incorporating a number of improvements we have figured out during the last six years of experimentation. However, from that point on the system will be frozen. We plan to continue using it in research, and hope others continue to find it useful as well, but our development efforts will be focused elsewhere. Some changes we hope to include in the final version include:

- *Dumper:* Creates a file that allows an envisionment to be reconstructed without recomputing it from scratch.
- *Syntax checker:* Flags syntax errors in domain models.
- *Domain-specific sums:* Being able to define sums over sets would greatly simplify many domain models. For example, one could define net-flow or net-force as the sum on the set of flows or forces affecting an object. Right now domain-specific sets are supported by QPE, but the necessary arithmetic procedures have not yet been integrated.
- *Batch mode:* While useful, the benchmark system isn't all that flexible. It would be useful to have software that lets one set up a machine as a "QPE server", which gets sent control, domain, and scenario files and which dumps envisionments where you specify.

References

[1] Collins, J. and Forbus, K. "Reasoning about fluids via molecular collections", Proceedings of AAAI-87, Seattle, Washington, July, 1987.

[2] Davis, E. "Order of magnitude reasoning in qualitative differential equations", Technical Report 312, NYU Computer Science Department, August, 1987

[3] de Kleer, J., Doyle, J., Steele, G., and Sussman, G. "Explicit control of reasoning" in Winston, P. & Brown, R., (Eds.), *Artificial Intelligence: An MIT Perspective: Volume 1*, The MIT Press, Cambridge, Mass, 1979

[4] de Kleer, J. "An assumption-based truth maintenance system", *Artificial Intelligence*, **28**, 1986

[5] de Kleer, J. and Brown, J. "A qualitative physics based on confluences", *Artificial Intelligence*, **24**, 1984

[6] Falkenhainer, B. "An examination of the third state in the analogy process: Verification-based analogical learning" Proceedings of IJCAI-87, August, 1987.

[7] Falkenhainer, B. and Forbus, K. "Setting up large-scale qualitative models" Proceedings of AAAI-88, August, 1988.

[8] Forbus, K. "Qualitative reasoning about physical processes" IJCAI-7, Vancouver, B.C., August, 1981

[9] Forbus, K. "Qualitative Process theory" MIT AI Lab Technical report No. 789, July, 1984.

[10] Forbus, K. "Qualitative process theory" *Artificial Intelligence*, **24**, 1984

[11] Forbus, K. "The Qualitative Process Engine", Technical Report No. UIUCDCS-R-86-1288, December, 1986.

[12] Forbus, K. "The logic of occurrence", Proceedings of IJCAI-87, Milan, Italy, 1987.

[13] Forbus, K. "`QPE`: A study in assumption-based truth maintenance" *International Journal of Artificial Intelligence in Engineering*, 1989.

[14] Forbus, K. "Introducing Actions into Qualitative Simulation" Department of Computer Science Technical Report No. UIUCDCS-R-88-1452, August, 1988.

[15] Forbus, K."Pushing the edge of the (QP) envelope", to appear in *Recent Progress in Qualitative Physics*, Faltings, B. and Struss, P. (Eds.), MIT Press, 1991.

[16] Forbus, K. and de Kleer, J. "Focusing the ATMS", Proceedings of AAAI-88, St. Paul, August, 1988.

[17] Forbus, K. and de Kleer, J. "The `ATMoSphere` Manual". In preperation.

[18] Hogge, J. "Compiling plan operators from domains expressed in Qualitative Process theory", Proceedings of AAAI-87, July, 1987.

[19] McAllester, D. "Reasoning Utility Package user's manual: Version one" MIT AI Lab Memo No. 667, April, 1982

[20] Siegle, G. "QPE interface manual", *in preparation.*

[21] Steele, G. *Common Lisp*, Digital Press, 1984.

[22] Weld, D. "Theories of comparative analysis", MIT AI Laboratory Technical Report No. 1035, May 1988.

A QPE files

This list is accurate as of September 23, 1988. However, it is subject to change without notice. For example, there will often be extra files containing subsystems being phased out or extra debugging routines.

careful Support for careful mode

cdi Support for defining processes and views.

chartop Character-oriented menu QPE interface.

cmenu Portable character-oriented menu system.

corres Correspondences.

debug Extra routines for debugging and consistency checking.

defs Basic QPE structures and sclass manipulation.

defsys QPE system definition.

domains Subdirectory of example domains.

efunct Explicit functions

examples Subdirectory of simple examples.

legal Legal notice (sigh!).

limit Limit analysis (including continuity tests and temporal inheritance).

pcvary Varying preconditions and modeling assumptions

psvs Builds first layer of state information and creates initial sclasses.

qdefs Core of quantity and inequality subsystem.

qlprint QPE-specific augmentations to the logic-pretty-printer in ATMoSphere.

qprint Report generator.

qrules ATMoSphere rules for bookkeeping associated with the quantity subsystem.

rel6 Subdirectory containing graphical interface for Symbolics Release 6.1.

rel7 Subdirectory containing graphical interface for Symbolics Release 7.2.

resolve Influence resolution code.

rt Subdirectory containing graphical interface for Lucid Common Lisp running on the IBM RT.

runs Subdirectory for storing simulation results. Generally comes with various test cases to check your installation.

symbols Describes what symbols from the user package are imported to the QPE package.

syntax Set of primitives which can be used in domain models. User-extendable.

syntaxer Data-driven syntaxer for expanding domain models into runnable `ATMoSphere` rules and assertions.

timer Simple portable timing utility.

top Procedures to be called by specialized interfaces and the benchmarking system.

trans Specialized inference system for transitivity reasoning.

Index

Allow-Coincidences, var, 15
Allow-Unstable-Trans, var, 14
Allow-discontinuous-directs, var, 15
Analysis-complete, var, 30
Assume-PCs, var, 14
Augment-Derivative-Comparisons, var, 14
Augment-Files, var, 31
BASIS, var, 31
Careful, var, 16
Case2-ECL-ON, var, 15
Domain-loaded, var, 30
Dump-situations, var, 16
ENFORCE-ENV, var, 31
ENVISION-ENV, var, 31
PS-VS, var, 31
PS-VS-Envs, var, 31
Prune-Asymptotic-Approaches, var, 15
QPE-benchmark-data-directory, var, 30
QPE-initialized, var, 30
QPE-rules-files, var, 30
Report-Mode, var, 16
Resolve-Simple-Multiple-Directs, var, 14
Scenario-loaded, var, 30
Silence-zero-Ds-values, var, 13, 16
benchmark-physics-files, var, 30
benchmark-scenarios, var, 19, 30
choice-sets, var, 29
differential-situation-display, var, 13
domain-expander-forms, var, 31
domain-files, var, 30
domain-path, var, 29
dump-situations, var, 19
examples-path, var, 29
explicit-procedures, var, 31
lisp-package, var, 29
named-choice-sets, var, 29
next-state-table, var, 29
qpe-pathname, var, 29
quantity-types, var, 31
scenario-files, var, 30
user-package, var, 29
user-report-hooks, var, 19
zero-struct, var, 29

+
    definition, 24
    QPE syntax, 24
++rel, QPE syntax, 24
+rel, QPE syntax, 24

-
    definition, 24
    QPE syntax, 24
--rel, QPE syntax, 24
->, QPE syntax, 23
-rel, QPE syntax, 25

:==, QPE syntax, 22

=, QPE syntax, 24

A, QPE syntax, 23
And, QPE syntax, 23
Asymptotic approach, 15
At-Most-One, QPE syntax, 23
adb::*debug-table*, var, 17
adb::core-dump, procedure, 11
adb::if-debug-flag, macro, 17
adb::set-debug-flag, procedure, 16
adb:assert!, procedure, 9
adb:assertq, macro, 9
add-recorder-event, macro, 16
addition, 24
amount?, procedure, 31
arithmetic, 24
asserting facts, 9

benchmark-qpe, procedure, 19
benchmarking QPE, 19

Choice-set, QPE syntax, 22
Choices, QPE syntax, 22
Correspondence
    definition, 24
    QPE syntax, 24
careful mode, 15
character-oriented interface, 10
choice sets, adding elements, 22
classes
    definition, 11

examining, 11
commands, 10
`consequences`, procedure, 11
continuity
direct influences, 15
switches, 14, 15
`contradiction-distribution`, procedure, 11
contradictions, examining, 11
conventions, 2
`count-sclasses`, procedure, 12
`count-situations`, procedure, 12

`D`, `QPE` syntax, 23
`DeclareAugment-File`, macro, 31
`DefAlgebraic-Operator`, macro, 25
`DefAugment-File`, macro, 31
`DefComputable-Relation`, macro, 21
`DefQPEmode`, macro, 30
`DefRelational-Operator`, macro, 25
`DefTranslation`, macro, 25
`debug-QPE-syntaxer`, procedure, 25
debugging, 16, 26
`defModelingAssumption`, macro, 22
`defModelingAssumptionType`, macro, 22
`defProcedure-name`, macro, 21
`defQuantity-Type`, macro, 21
`defQuantityType`, `QPE` syntax, 23
`defUser-Report-Generator`, macro, 19
`defentity`, macro, 22
`defpredicate`, macro, 22
`derivative?`, procedure, 31
direct influences, displaying, 12
domain models, constructing, 21
`dump-envisioner-results`, procedure, 18

Event scheduler, 19
`Exists`, `QPE` syntax, 25
`e`, procedure, 10
`env-sclass`, procedure, 13
`env-sclass?`, procedure, 13
environments
as situations, 13
displaying, 11
indexing, 10
`envision`, procedure, 30
`equal-to`, `QPE` syntax, 24
`event-breakdown`, procedure, 16
`expand-scenario`, procedure, 30

fact, examining, 11
facts, examining, 11
flags, 16
flight recorder, 15

`gather-preconditions`, procedure, 30
`gather-quantities`, procedure, 30
`get-individuals`, procedure, 30
`get-ineq-prop`, procedure, 30
`get-lh-prop`, procedure, 30
`get-pis`, procedure, 30
`get-qnum-prop`, procedure, 31
`get-sclass-prop`, procedure, 30
`get-vis`, procedure, 30
`greater-than`, `QPE` syntax, 24

`I+`, `QPE` syntax, 24
`I-`, `QPE` syntax, 24
Influence resolution, controlling, 14
`Introduces`, `QPE` syntax, 25
`iff`, `QPE` syntax, 23
indirect influences, displaying, 12
`ineq`, struct, 29
`ineq-num1`, procedure, 31
`ineq-num2`, procedure, 31
inequalities, displaying, 12
influence resolution, mode switches, 14
inputs to `QPE`, 9
installation, 6
interfacing `QPE` to other systems, 29
interrogatives, 10

`justifications`, procedure, 11

legal restrictions, 4
`less-than`, `QPE` syntax, 24
`lh`, struct, 29
`lh-stats`, procedure, 13
limit analysis, switches, 14, 15
limit hypotheses
examining, 13
listing, 13
`limit-analysis`, procedure, 30
`limit-hypothesis?`, procedure, 29
`list-all-sclasses`, procedure, 30
`list-events`, procedure, 16
listings, 18
`load-domain`, procedure, 30
`load-scenario`, procedure, 30
`lookup-sclass`, procedure, 30

`m`, QPE syntax, 23
`make-amount`, procedure, 30
`make-derivative`, procedure, 30
`map-sclasses`, procedure, 30
`map-situations`, procedure, 30
menu system, starting, 10
mode switches, preconditions, 14
modes
    definition, 13
    displaying, 12

`Not`, QPE syntax, 23
numbers, displaying, 12

`Only-During`, QPE syntax, 25
`Or`, QPE syntax, 23
operating QPE, 9
QPE overview, 1

Printing, controlling, 13
`Procedure-Spec`
    definition, 24
    QPE syntax, 24
packages, 3
preconditions, assuming or asserting, 14
`princ-lh`, procedure, 13
`print-QPE-modes`, procedure, 12
`print-contradictions`, procedure, 11
`print-ds-values`, procedure, 13
`print-env`, procedure, 11
`print-envs`, procedure, 11
`print-lh`, procedure, 13
`print-next-state-table`, procedure, 13
`print-sclass`, procedure, 12
`print-situation`, procedure, 13
`print-situation-lh-table`, procedure, 13
`print-status`, macro, 11
procedures
    anonymous, 24
    named, 24
    specifying additional information, 24
process instances, displaying, 11
`put-ineq-prop`, procedure, 30
`put-lh-prop`, procedure, 30
`put-qnum-prop`, procedure, 31
`put-sclass-prop`, procedure, 30

`Q=`
    definition, 24
    QPE syntax, 24
`QP-inequality?`, procedure, 29
`QP-number?`, procedure, 29
`QP-quantity?`, procedure, 29
`Qprop`, QPE syntax, 24
`Qprop-`, QPE syntax, 24
`Quantity`, QPE syntax, 23
`qnum`, struct, 29
`qpe`, procedure, 10
qualitative proportionality, 24
`quant`, struct, 29
quantities, displaying, 12
quantity, type declaration, 23
quantity space, displaying, 12
`quantity-of`, procedure, 30
`quantity-type?`, procedure, 30

`Rule`, QPE syntax, 23
`reload-QPE-rules`, procedure, 30
`reload-augment-files`, procedure, 31
report generation
    definition, 18
    switches, 16
`rerun-qpe`, procedure, 19

`Set`, QPE syntax, 25
`Set-Model-Assumption`, macro, 9
`Settings`, QPE syntax, 22
`Switch`, QPE syntax, 22
`s`, QPE syntax, 23
scenario expansion, displaying, 11
`schedule-qpe-event`, procedure, 20
`sclass`, struct, 29
`sclass-r-completions`, procedure, 13
`sclass?`, procedure, 29
sclasses
    definition, 12
    examining, 12
    number of, 12

r-completions of, 13
`show-cdis`, procedure, 11
`show-choice-sets`, procedure, 12
`show-class`, procedure, 11
`show-classes`, procedure, 11
`show-data`, procedure, 11
`show-events`, procedure, 16
`show-ineq`, procedure, 12
`show-lhs`, procedure, 13
`show-number`, procedure, 12
`show-pis`, procedure, 11
`show-possible-directs`, procedure, 12
`show-possible-indirects`, procedure, 12
`show-qpe-internal-events`, procedure, 20
`show-qstate`, procedure, 12
`show-quantities`, procedure, 12
`show-vis`, procedure, 11
situations
definition, 12
examining, 13
limit hypotheses of, 13
number of, 12
`special-quantity?`, procedure, 31
`start-qpe`, procedure, 30
statistics
flight recorder, 16
of envisionments, 12
of limit hypotheses, 13
subtraction, 24
`summarize-Ds-values`, procedure, 12
`summarize-qpe-statistics`, procedure, 12
switches, distinction between flags and modes, 13

`Taxonomy`, `QPE` syntax, 23
`There-Is`, `QPE` syntax, 25
`There-Is-Unique`, `QPE` syntax, 25
transitions, summary of, 13
`types-of-events`, procedure, 16

`Uniquely-Introduces`, `QPE` syntax, 25

`Ds` values, distribution of, 12
view instances, displaying, 11
`view-qnum`, procedure, 31

`When`, `QPE` syntax, 23
`When-Careful`, macro, 16
`why`, procedure, 11
`whyq`, procedure, 11