SANDIA REPORT

SAND2018-6660
Unlimited Release
Printed June 2018

How to ADAPT

Zachary Jankovsky, Troy Haskin, Matthew Denman

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy

Office of Scientific and Technical Information

P.O. Box 62

Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Available to the public from

U.S. Department of Commerce

National Technical Information Service

5285 Port Royal Rd

Springfield, VA 22161

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2018-6660
Unlimited Release
Printed June 2018

How to ADAPT

Zachary Jankovsky, Troy Haskin, Matthew Denman

Risk & Reliability Analysis

Sandia National Laboratories

P.O. Box 5800

Albuquerque, NM 87185-MS0748

Abstract

The ADAPT software allows for the examination of aleatory and epistemic uncertainties in complex system transients using the Dynamic Event Tree (DET) methodology. This manual outlines the principles of operation of ADAPT and provides directions for its use. Future plans for the code are briefly outlined.

Acknowledgments

The authors wish to acknowledge Emily Sandt and Bibiana Seng for their reviews of this document and recommendations for improvements.

This work was largely funded through internal Sandia National Laboratories division support. Incidental feature-specific content was produced under CRADA SC18/1904.00 with Oklo Inc.

Contents

Figures

Tables

Listings

Nomenclature

ADS Accident Dynamics Simulator

APET Accident Progression Event Tree

BRF Branching Rules File

CDF Cumulative Distribution Function

DET Dynamic Event Tree

DPRA Dynamic Probabilistic Risk Assessment

DYI Dynamic Importance

ET Event Tree

ET/FT Event-Tree/Fault-Tree

FT Fault Tree

HPC High Performance Computing

ISLOCA Interfacing System Loss of Coolant Accident

LDRD Laboratory Directed Research and Development

LOFC Loss of Forced Circulation

MAAP4 Modular Accident Analysis Program 4

NPP Nuclear Power Plant

OSU The Ohio State University

PBMR Pebble Bed Modular Reactor

PRA Probabilistic Risk Assessment

PWR Pressurized Water Reactor

RELAP5 Reactor Excursion and Leak Analysis Program 5

SBO Station Blackout

SCP Secure Copy

SFR Sodium-cooled Fast Reactor

SNF Spent Nuclear Fuel

SNL Sandia National Laboratories

SQL Structured Query Language

SSH Secure Shell

TSIF Template Simulator Input File

UTOP Unprotected Transient Overpower

1 Introduction

ADAPT[1] is a piece of software that generates and analyzes Dynamic Event Trees (DETs) to evaluate aleatory and epistemic uncertainties for Dynamic Probabilistic Risk Assessment (DPRA). ADAPT was developed under Sandia National Laboratories (SNL) Laboratory Directed Research and Development (LDRD) project 79780 in cooperation with The Ohio State University (OSU) [1]. ADAPT has been applied to a number of accident scenarios for different nuclear reactor types as well as non-reactor systems. Broadly, ADAPT can be applied in any case where

- there is a complex system problem that may be represented by an event tree,
- there is a computer code that can capture the likely progression of events, and
- that computer code may be made to stop after designated events occur and restart the analysis with changed input.

This manual summarizes the method of operation of ADAPT and gives examples of scenarios that have been analyzed using the code. Also included is a list of references for further reading on previous analyses. The remaining sections of the manual are arranged as follows:

- Section 2 describes the user interface of ADAPT and walks through how a simulator may be added and an experiment started.
- Section 3 gives an example of the process of creating ADAPT input.
- Section 4 describes how outputs are gathered from individual sequences and analyzed as a whole.
- Section 5 concludes the manual with a brief description of future plans for ADAPT and provides contact information for code requests.

Beyond the main body of the manual are a number of appendices which list the user-accessible modules and functions in ADAPT as well as give complete input for the sample problem used throughout the manual:

- Appendix A describes significant executable ADAPT modules.
- Appendix B lists significant ADAPT functions.
- Appendix C gives a sample ADAPT wrapper file.
- Appendix D gives a sample ADAPT Branching Rules File (BRF).

[1] ADAPT is no longer an acronym.

- Appendix E gives a sample ADAPT Template Simulator Input File (TSIF).
- Appendix F walks through an analysis of the sample problem using ADAPT tools.

This chapter serves as an introduction to ADAPT including a short background on DPRA. Section 1.1 defines some specific terms that are used throughout this manual. Section 1.2 gives an overview of traditional Probabilistic Risk Assessment (PRA). Section 1.3 describes DPRA and the DET approach in particular. Section 1.4 outlines the structure of ADAPT as well as the history of the project. Within this manual, the names of files are given in *italics*. Modules, functions, and commands are given in ***bold italics***. Variables within the code are given in a `fixed-width font`.

1.1 Definitions

Certain terms are used in this manual which have a specific meaning within the context of ADAPT or dynamic PRA in general.

Branch: a segment of the analysis with a set of uncertain system parameters that remain constant until a branching condition is reached

Checkpoint: to cause a simulator to close in a way that preserves progress and allows continuation at a later time

End State: the branch at which an analysis ending condition is reached

Experiment: a dynamic event tree generated by ADAPT

Job: an attempt to run the input associated with a branch on a particular computation host

Sequence: the unique chain of branches from the initiating event to an end state

Simulator: a piece of software that models a physical system

Simulator Package: a set of files that define how ADAPT shall run an experiment

1.2 Probabilistic Risk Assessment

PRA is an analytic tool that seeks answers to three questions:

- What failure events might affect the system?
- What are the possible consequences of an initiating failure event?
- What is the likelihood of each set of consequences?

PRA uses Event-Tree/Fault-Tree (ET/FT) analysis to evaluate the impacts of internal or external initiating events on a complex system. In a Fault Tree (FT), basic failure events are assembled using primarily AND/OR logic to determine the combinations of individual component failures that lead to failure of the system as a whole (a "top event"). Fault trees are generated for top events that represent a system failing to meet one or more of its design goals. For example, in Nuclear Power Plant (NPP) PRA a top event may be "Loss of Electric Power to Engineered Safety Features" [2] as seen in Figure 1. These fault trees are stored in a database and can be used for a variety of analyses.

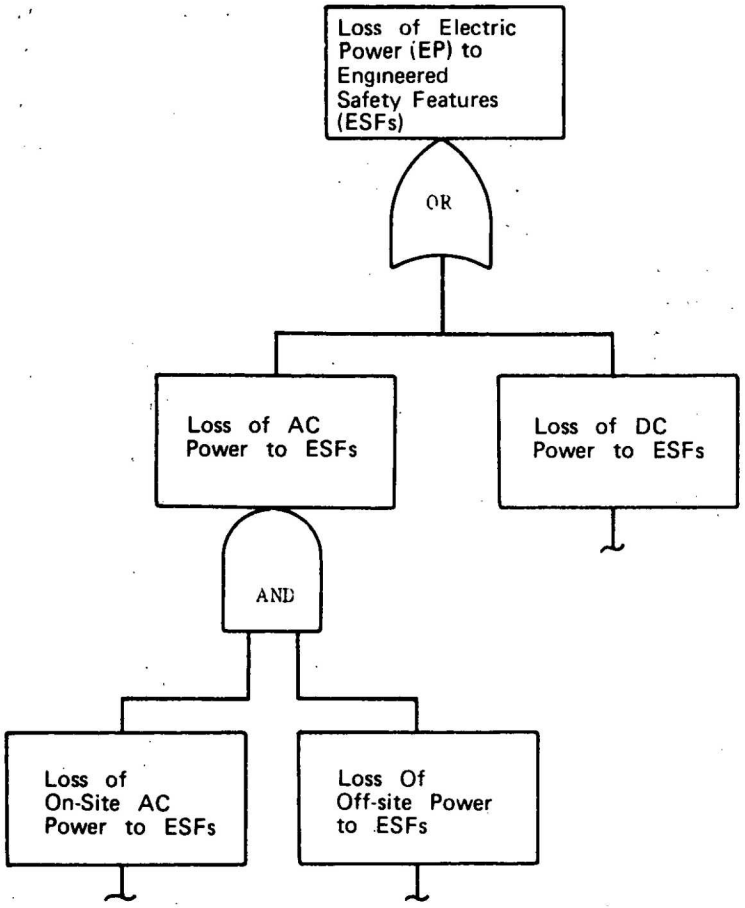

Figure 1: Illustration of a Fault Tree [2]

An Event Tree (ET) is forward-facing and begins with a single initiating event. An example of an initiating event is "Small to intermediate pipe breaks (2" to 6" equivalent diameters)" [2] as seen in Figure 2. From there the analysis considers what event may occur next (based on analyst knowledge of the system) and branches out among the possible success and failure of a given event. Each of these intermediate events (e.g., *Electric Power* in Figure 2) represents the top event of a fault tree. This branching continues until user-defined end states are realized. End states may include a safe and stable configuration of the system or any number of differing failure states. Both fault trees and event trees require basic event probabilities as inputs to provide insight on the likelihood of different outcomes.

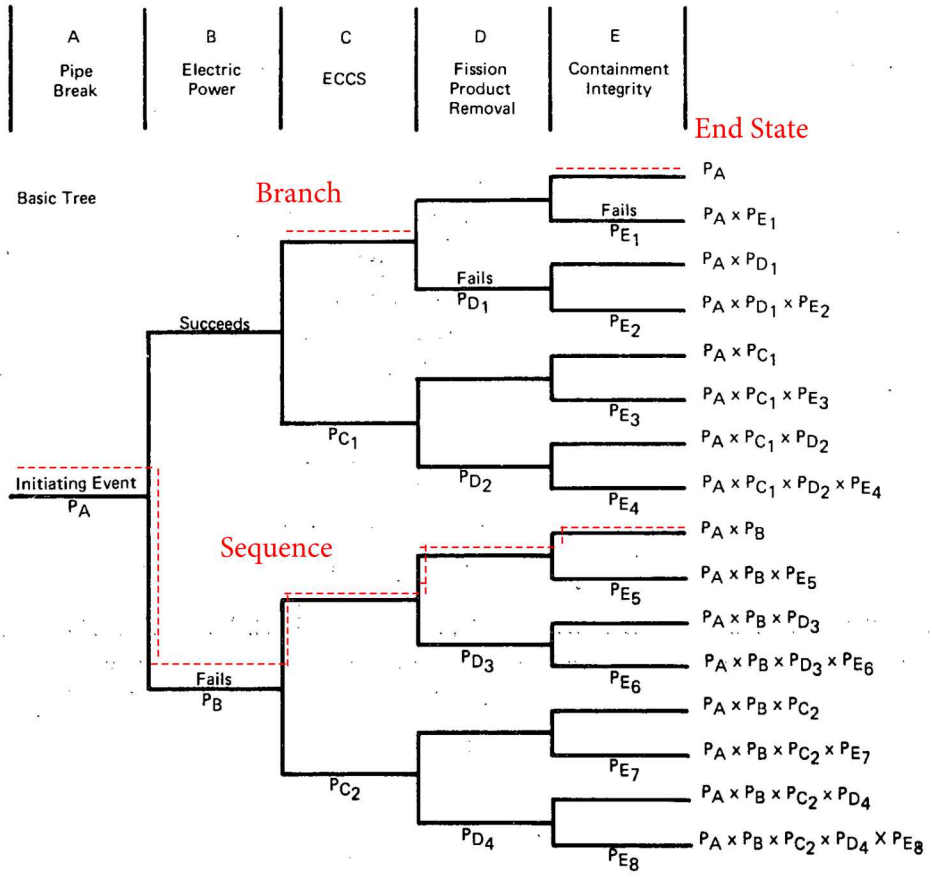

Figure 2: Simplified Event Tree for a Loss of Coolant Accident [2]

The end result of a PRA is quantification of the risk of the overall system failing due to a given initiating event. The use of PRA has increased in recent decades as computational power has made it feasible to represent more aspects of complex systems including structural, hardware, software, and human reliability effects. PRA is not discipline specific and has been used in industries such as nuclear power [2, 3, 4], oil and gas extraction [5], and aerospace [6].

1.3 Dynamic PRA

Two limitations of traditional ET/FT PRA are that the order of events is determined by the analyst a priori and that the effects of event timing are not explicitly captured [7]. The lack of a timing consideration also affects the handling of logic loops in which a system may transition between states multiple times depending on physical conditions. While there are numerous DPRA methodologies that attempt to solve these weaknesses, ADAPT uses the DET approach as shown in Figure 3. One advantage of the DET approach is its direct integrability into existing ET/FT PRA [7]. Subjectivity in the ordering of events is reduced under the DET approach by using the output of a dynamic system model (simulator) to inform the branching. Branching conditions are triggered by the existence of a relevant system state in the simulator and therefore only occur when physically appropriate. The DET approach also allows consideration of both epistemic and

aleatory uncertainties on a phenomenologically and stochastically consistent platform. It is important to note that DETs represent a departure from traditional ETs in that branching in DETs is based on time-dependent behavior represented by a software simulator while in traditional ETs branching is based on manual decisions by the risk analyst.

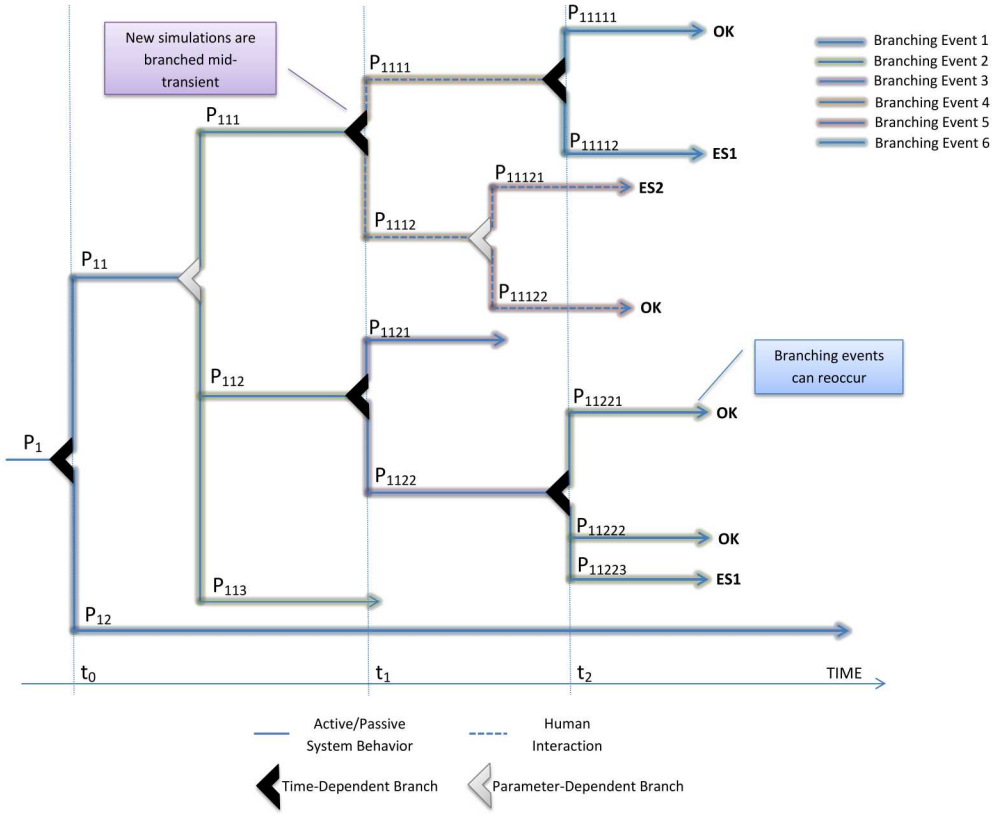

Figure 3: Diagram of DET Branching [8]

The input for a DET consists of a set of branching conditions that are applied to a computer model of a system. For example, in an NPP DET a branching condition may be to "determine piping pressure capacity upon overpressurization." First, the simulator must be made to stop when the pressure exceeds a pre-set value that represents imminent overpressurization. For the ADAPT DET driver, the simulator is expected to be programed to do this[2]. In simulators such as MELCOR, this behavior is programed using the Control Function module [9]. Similar modules exist in most other system-level nuclear safety analysis codes. The DET driver must be informed of the exact branching condition that was reached.

Next, the DET driver reads a database entry for the appropriate branching condition. This entry contains conditional probabilities and the changes to make to the simulator input file for each new branch to be created. In the example of determining piping pressure capacity, a Cumulative Distribution Function (CDF) of the pipe's capacity would be sampled according to some adaptive or pre-defined binning scheme to address epistemic uncertainty in the pipe's strength. The DET driver will create a number of new branches in a database each with a conditional probability and

[2]Some DET driver codes, most notably Accident Dynamics Simulator (ADS), are tightly coupled to the simulator and can query physical states through shared memory.

a set of changes to make to the simulator input file. The new branches are sent to a queue where they wait until computational resources are available. The process ends when no new branches are waiting to be run, typically when all sequences have reached the end of the simulation time of interest. An important corollary is that the maximum simulation time must be chosen such that the effects of phenomena of interest will be captured. Finding an appropriate maximum simulation time may require the use of small-scale pilot simulations run manually or as a DET.

1.4 ADAPT Overview

ADAPT is a collection of scripts and functions that are used to schedule jobs and report results to a central database. Some general features of ADAPT, both as a piece of software and a project, are outlined in Section 1.4.1. ADAPT was created to be flexible in terms of the simulators that may be linked. To that end, ADAPT has been designed to have a loose coupling to the simulator [1, 10]. This design decision has led to it being linked to a number of simulators as seen in Table 1. Another benefit of this method is that the effect of a simulator failure is limited to the immediate branch that was being run [11]. Nuclear safety simulator codes combine numerous physical models and, in uncertainty analyses, sets of input parameters may be encountered for which the simulator fails to converge [12]. In such cases a tight coupling of the DET driver and the simulator may result in difficulty completing the DET [11].

The basic requirements for linking a simulator to ADAPT are given in Section 1.4.2. ADAPT allows flexibility in the arrangement of computational hosts as noted in Section 1.4.3. Section 1.4.4 provides a brief reminder of the importance of managing scope when generating DETs. Section 1.4.5 tabulates publications relating to the development of ADAPT as well as specific analyses that have been performed.

1.4.1 General Code Features

ADAPT is written in the Python programming language (minimum 2.7) and consists of a number of executable scripts as well as user-accessible libraries of functions. As of the time of publication it has been used under Red Hat type Linux systems and macOS[3]. It was recently revised for cross-compatibility between Python 2 and 3 as well as to remove operating system dependent functions in preparation for a Windows version [13].

ADAPT was put under revision control using Subversion at SNL in December 2015. As of the time of publication approximately 400 sets of changes have been committed along with descriptions of the motivation of the change. The repository includes a test case (using MELCOR [14]) as well as template ADAPT configuration files for a number of common computational arrangements.

1.4.2 Simulator Requirements

The basic requirements for linking a simulator to ADAPT, which are similar to those for other DET driver codes, are that the simulator must:

[3] In this manual, Linux is used generically to refer to Unix-like operating systems.

- accept modified input parameters for its control system upon restart of analysis,
- stop on control system values crossing a pre-defined threshold,
- output the reason for any code stoppage, and
- stop when commanded by ADAPT.

The MELCOR code gained these capabilities shortly before ADAPT was created [15] and was the first code linked to ADAPT [1]. The process of modifying an existing code to meet the requirements of ADAPT is described in Reference [16]. The simulator codes linked to ADAPT at the time of publication are listed in Section 1.4.5.

1.4.3 Computational Arrangement

The typical computational arrangement for ADAPT is diagramed in Figure 4. The web interface is accessible from any operating system. Through the web interface, the user can control the progress of experiments as well as view basic results. In each application thus far, ADAPT has been run on a computer cluster running Linux using Secure Shell (SSH) to facilitate communication between the head node and computation nodes. In cluster arrangements it is typical for the head node to be the host of the job server (***adapt-server***), web server (***adapt-webmin***), and Structured Query Language (SQL) database which is currently implemented in MySQL. These services do not necessarily have to be run on the same machine[4].

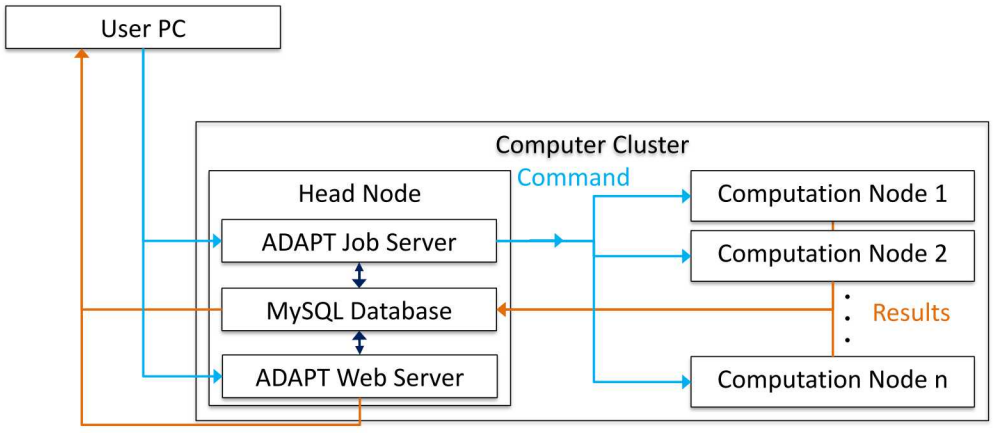

Figure 4: ADAPT Computational Arrangement

ADAPT can use any machine it can reach via an SSH connection as a computation node[5]. The head node and computation hosts have been Linux machines thus far, but a Windows version of ADAPT is in development. The details of running the ADAPT services are given in Section 2.

[4]It is feasible to run the database, the servers, and computations on the same desktop machine.

[5]The simulator executable must be capable of running on each desired host machine, which requires the resolution of prerequisites such as licensing and the availability of code libraries.

1.4.4 A Note on Scoping Branching Rules

Traditional PRAs are limited by the capability of human analysts to apply their knowledge of accident progression to create new sequences. It is common to bin similar sequences when analyzing outputs in large-scale analyses. Additionally, sequences of particularly low calculated consequences may be pruned from the final analysis [4]. In a DET analysis, branching rules are input and will occur as often as physically appropriate. As such, it is difficult to predict the eventual size of a DET and it is easy to create input for a DET that will not finish in a reasonable time with the available computational resources. Previous DET analyses have commented on the scale of analyses:

- Hakobyan (2006): "In a typical Accident Progression Event Tree (APET), the number of branches may reach several hundreds or even thousands." [17]
- Metzroth (2011): "The full calculation of the APET can produce hundreds of thousands of scenarios which is extremely cumbersome to post-process without some method of filtering the results." [18]
- Brunett (2013): "Through the use of the ADAPT DET generation technique, the amount of data generated and size of trees produced is manageable, and the simulation runtimes are not overly cumbersome if coupled with a relatively fast running simulator such as MELCOR." [19]
- Osborn (2013): "It took approximately 4 months to download and convert all the Test Case 1 ADAPT/MELCOR outputs." [20]
- Jankovsky (2018): "Currently, 1,448,618 branches have been identified (representing 781,763 sequences) and 697,663 branches have finished running." [21]

Numerous approaches exist to limit or manage the data produced by a DET. Most DET driver codes allow the user to enter a cutoff probability below which new branches will not be run [17]. The effect of this threshold can be significant as each terminated branch may have resulted in many end states. After the DET has finished, the data may be reduced by binning similar sequences [18, 19, 20]. This approach does not save computation time but may retain more of the variability of outputs of the DET in comparison to a probability threshold.

1.4.5 ADAPT Publications

Previous analyses using ADAPT are listed in Table 1[6]. Related analyses are combined to the best of the authors' knowledge and entries are arranged in order of date of first publication. Publications focused more on the development of ADAPT than on any particular analysis are listed in Table 2 along with a short description of the development work.

Table 1: Published Analyses using ADAPT

Years	Reactor Type	Accident Type	Simulator	Reference
2006-2011	PWR	SBO	MELCOR	[17, 22, 23, 24, 10, 18]
2009	SFR	Aircraft Crash	RELAP5	[25]
2013	PWR	SBO	MELCOR	[20, 26]
2013-2014	PWR	SBO	MELCOR	[19, 27]
2014	PBMR	LOFC	MELCOR	[28]
2015-2017	PWR	SBO	MAAP4	[29]
2015-2017	SFR	UTOP	SAS4A	[8, 16, 30]
2015-2018	PWR	ISLOCA	MELCOR	[31, 32]
2015-2018	BWR	SBO	MELCOR	In Progress
2016-2017	N/A	SNF Transport	Multiple	[33]

Table 2: Other Publications relating to ADAPT

Year	Description	Reference
2006	ADAPT Software Overview	[34]
2006	Visualization of Event Trees	[35]
2008	Initial ADAPT LDRD Closeout	[1]
2016	Introduce dynamic importance measures	[36]
2016	Introduce conditional tree reduction	[37]
2016	Introduce multi-simulator operation	[38]
2016	General improvements presentation	[13]

[6]A number of terms appear in Table 1 that are not used elsewhere: Pressurized Water Reactor (PWR), Station Blackout (SBO), Sodium-cooled Fast Reactor (SFR), Reactor Excursion and Leak Analysis Program 5 (RELAP5), Pebble Bed Modular Reactor (PBMR), Loss of Forced Circulation (LOFC), Modular Accident Analysis Program 4 (MAAP4), Unprotected Transient Overpower (UTOP), Interfacing System Loss of Coolant Accident (ISLOCA), and Spent Nuclear Fuel (SNF).

2 Running ADAPT

This section describes how the user interacts with ADAPT to run and check the progress of cases. Section 2.1 provides template information for the ADAPT configuration files. Section 2.2 describes the commands required to start and stop the long-running ADAPT modules. Finally, Section 2.3 walks through the steps necessary to add a simulator and run an experiment using the web interface.

2.1 Configuration Files

Two configuration files (*.bashrc* and *.adaptrc*) are required to define ADAPT's general operation. The first is *.bashrc* or the equivalent for the shell being used. In addition to the user's existing *.bashrc* commands, the content in Listing 1 must appear in order to locate ADAPT executables and the second configuration file (*.adaptrc*).

Listing 1: Modifications to .bashrc for ADAPT

```
1 PATH=$PATH:~/ADAPT/ADAPT/server
2 PATH=$PATH:~/ADAPT/ADAPT/scripts
3 ADAPTRC=~/ADAPT/ADAPT
4 export PATH
5 export ADAPTRC
```

The content in Listing 1 assumes that the ADAPT repository (see Section 1.4.1) has been copied to the user's home directory which is represented by ~. The *server* and *scripts* directories contain the ADAPT modules and common post-processing tools, respectively, and are added to the user's `PATH` environment variable which defines locations to search for executable files (lines 1 and 2). The location of the ADAPT configuration file (*.adaptrc*) is set as environment variable `ADAPTRC` (line 3). The `PATH` and `ADAPTRC` variables are made persistent using the ***export*** command (lines 4 and 5).

Listing 2 shows the first section of *.adaptrc* which defines the location of temporary file directories and log files. `server_webmin` defines the location to store files uploaded through the web interface (line 2). `server_adaptemp` defines the location of temporary files used to process finished branches and create new ones (line 3). Files in `server_adaptemp` are not needed permanently and are purged once no longer necessary. `server_log` and `server_logwebmin` define the locations of log files for ***adapt-server*** and ***adapt-webmin***, respectively (lines 4 and 5).

Listing 2: .adaptrc configuration file, server section

```
1 [server]
2 server_webmin = ~/ADAPT/ADAPT/data/webmin
3 server_adaptemp = ~/ADAPT/ADAPT/data/adaptemp
4 server_log = ~/ADAPT/ADAPT/data/adapt-server.log
5 server_logwebmin = ~/ADAPT/ADAPT/data/adapt-webmin.log
```

The configuration section given in Listing 3 defines the SQL database to be used. The database does not need to be running on the same machine as ***adapt-server*** or ***adapt-webmin***[7]. `database_host` (line 2 in Listing 3) and `database_port` (line 3) define the location and port to use to access the SQL database. Creation of the SQL database and user generally must be coordinated with the system administrator. This task may be automated with the module ***adapt-database-admin*** which is described in Appendix A. ADAPT is flexible with regard to the database structure and if multiple users exist, they may share a database or each have their own. `database_db` (line 4) defines the name of the database to use for ADAPT. `database_user` (line 5) and `database_pass` (line 6) define the username and password to access the database.

Listing 3: .adaptrc configuration file, database section

```
1 [database]
2 database_host = localhost
3 database_port = 3306
4 database_db = adapt_db
5 database_user = adapt_user
6 database_pass = adapt_pw
```

The section of *.adaptrc* in Listing 4 defines the commands to use to run system commands and copy files, the computation hosts to use, and the directories to use for output data. The syntax `ssh -x -o StrictHostKeyChecking=no` (line 2 in Listing 4) is the default used by ADAPT to execute commands on remote hosts using SSH. The options ensure a consistent and streamlined connection across different hosts. The syntax `scp -q -o StrictHostKeyChecking=no` (line 3) is used by ADAPT to copy files between file systems using Secure Copy (SCP). `ssh_hosts` (line 4) defines a list of hosts for execution of jobs. Computation host names are repeated for each execution slot desired on the host and so the input in Listing 4 will allow up to three branches to run simultaneously on `localhost`. `ssh_directory` (line 5) defines the data storage location for each unique host in `ssh_hosts`. Regular expressions may be used for matching. For example, `^local:~/data` would match any host whose name starts with `local` to the directory `~/data`.

Listing 4: .adaptrc configuration file, ssh section

```
1 [ssh]
2 ssh_ssh = ssh -x -o StrictHostKeyChecking=no
3 ssh_scp = scp -q -o StrictHostKeyChecking=no
4 ssh_hosts=localhost localhost localhost
5 ssh_directory = localhost:~/ADAPT/ADAPT/data/outputs
```

Finally, Listing 5 shows the configuration of the web interface. `webmin_users` (line 2 in Listing 5) defines pairs of usernames and passwords for authorization to use the web interface. `webmin_port` (line 3) defines the port to use for the web interface. Generally, the port must be opened in the system firewall by an administrator before it can be accessed remotely. `webmin_baseurl` (line 4) defines the address that will be used to define paths within the web interface.

[7]Some organizations prefer to host databases on centralized and/or purpose-selected hardware.

Listing 5: .adaptrc configuration file, webmin section

```
1 [webmin]
2 webmin_users = user1:pass1 user2:pass2
3 webmin_port = 31415
4 webmin_baseurl = http://127.0.0.1:31415
```

2.2 Starting and Stopping ADAPT

The long-running processes of ADAPT must be commanded to start and stop via the command line and may be done through an SSH connection to the machine that is intended to host ***adapt-server*** and ***adapt-webmin*** (see Figure 4). The command ***adapt-ops start adapt*** starts the long-running ADAPT job server as seen in Listing 6. Likewise, as seen in Listing 7 ***adapt-ops stop adapt*** commands the ADAPT job server to shut down. A token is inserted into the SQL database which the job server recognizes as an instruction to close. The ***adapt-server*** process must be running for ADAPT to prepare and launch jobs to complete branches.

Listing 6: adapt-server start command

```
1 [user@host ~]$ adapt-ops start adapt
2 Started adapt-server at Tue Feb 7 09:43:37 2017.
```

Listing 7: adapt-server stop command

```
1 [user@host ~]$ adapt-ops stop adapt
2 Stopped adapt-server using adapt-server-halt at Tue Feb 7
    09:45:12 2017.
```

The ADAPT web interface is contained within ***adapt-webmin***. Many user interactions with ADAPT may be performed through the web interface, although all commands may also be issued by the command line for ease of automation. ***adapt-webmin*** is started and stopped using ***adapt-ops***: ***adapt-ops start webmin*** and ***adapt-ops stop webmin*** start and stop it as shown in Listings 8 and 9, respectively. The ***adapt-webmin*** process must be running to use the web interface but is not necessary for jobs to run.

Listing 8: adapt-webmin start command

```
1 [user@host ~]$ adapt-ops start webmin
2 Started adapt-webmin at Tue Feb 7 09:43:45 2017.
```

Listing 9: adapt-webmin stop command

```
1 [user@host ~]$ adapt-ops stop webmin
2 Stopped adapt-webmin at Tue Feb 7 09:45:17 2017.
```

2.3 Using the Web Interface

The web interface is the primary method of interaction with ADAPT. It contains a number of menu options as shown in Figure 5 which allow the user to create and utilize simulator packages for new experiments. In ADAPT, a simulator package is a set of files that are required to generate a DET. This package includes a simulator executable which is commonly uploaded at the time of simulator package creation as well as a list of input files that must be uploaded for each new DET. An experiment (or case) is a set of branching rules and simulator input that can be run under a simulator package to create a new DET.

In order to access the web interface, the user needs to connect to the host via a web browser and uses the user name and password as specified in *.adaptrc* (see Listing 5). Only menu options 1, 2, and 3 are examined in this brief introduction in Sections 2.3.3, 2.3.2, and 2.3.1, respectively. These tasks are presented in the expected running order: add a simulator package, launch an experiment, and check its progress.

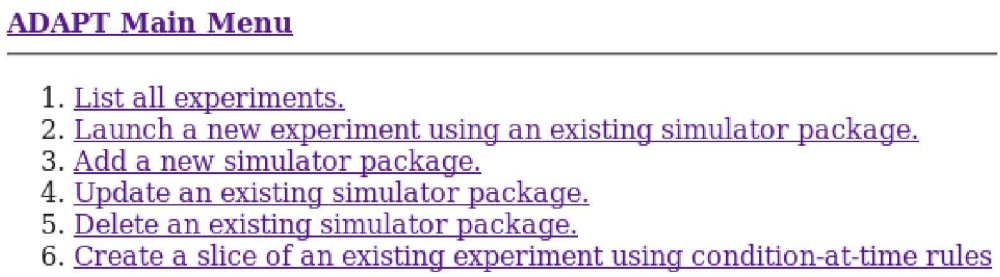

Figure 5: ADAPT Main Menu

2.3.1 Adding a Simulator

If the user chooses to add a simulator package, a page similar to Figure 6 is displayed. The simulator package is named and files are defined as required. The web wrapper is only required when using the web interface as it defines how the web interface is to submit the first branch to the database. The main wrapper is required regardless of the method of submission. Clicking on ***Choose File*** (Chrome)[8] or ***Browse*** (Firefox) allows the user to navigate the local filesystem for the appropriate file. The names of files may be defined by the user for all but the web wrapper and the main wrapper. A typical set of files to be uploaded for a simulator package and an experiment are given in Section 3.2.

[8]The ADAPT web interface is browser-independent but the file selection button `input type=file` is rendered differently in different browsers.

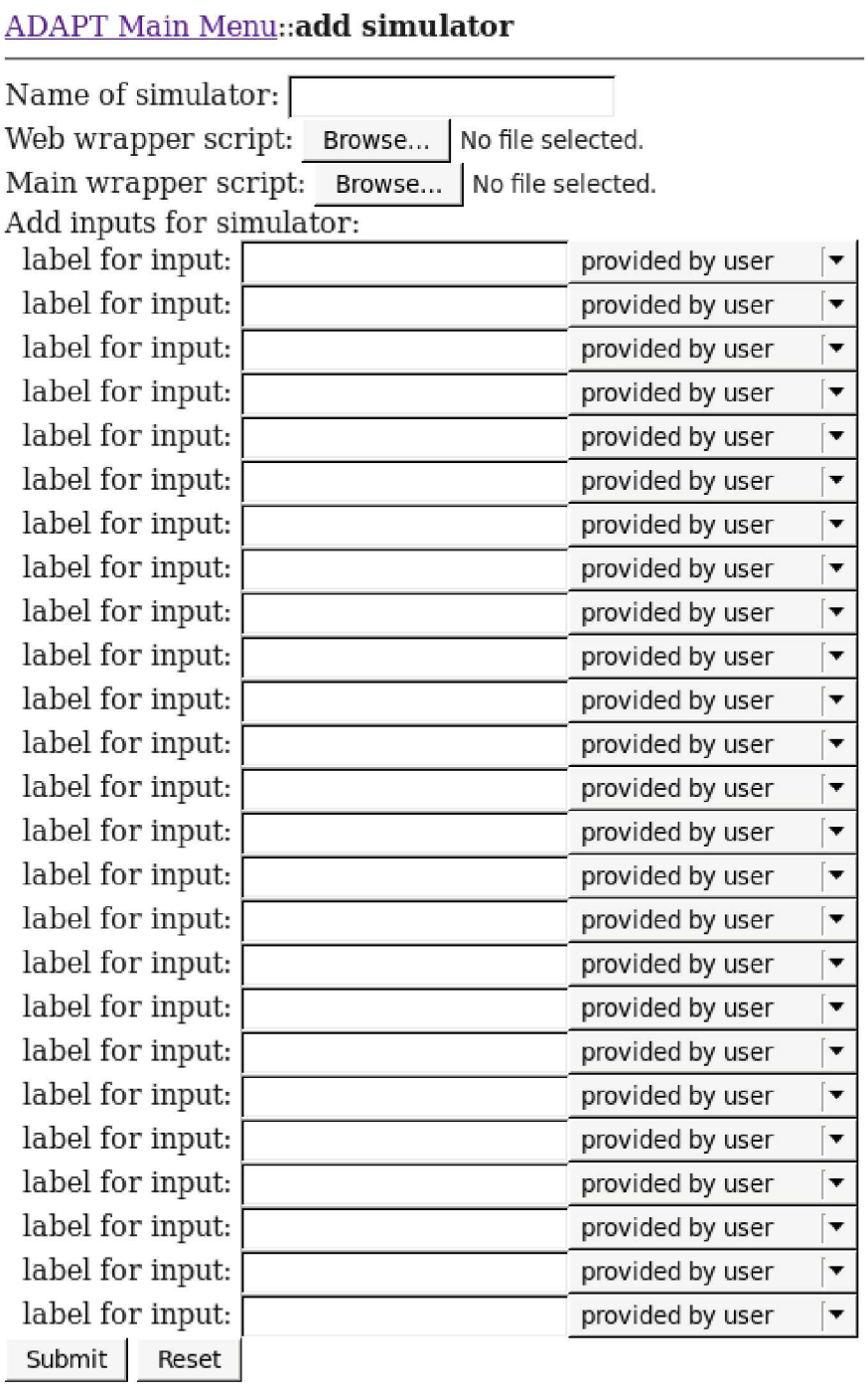

Figure 6: ADAPT Add Simulator

Next, names are given by the user for other simulator input files. Files that are specific to the experiment (such as the TSIF, BRF, and restart file) use the type `provided by user` from the drop-down menu. Files that do not change between experiments with the same simulator package, such as the simulator executable, use the type `provided by installer` and are uploaded at the point of creating the simulator package. Files that are `provided by user` are uploaded at the time of launching an experiment (see Section 2.3.2). Figure 7 shows a fully-defined set of input for an ADAPT simulator package.

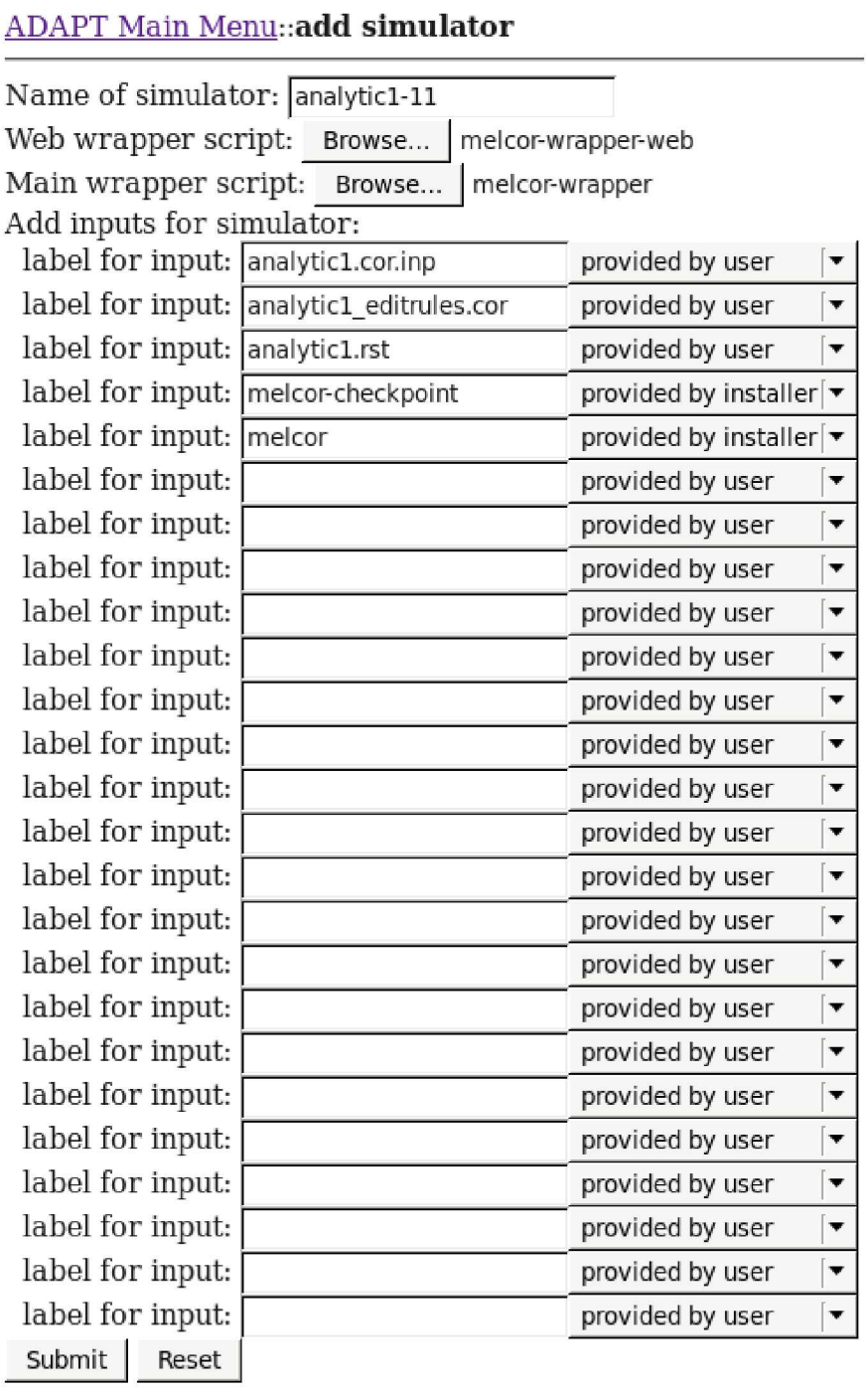

Figure 7: ADAPT Add Simulator: Files Defined

A list of the required files for an ADAPT experiment is given in Section 3.2. If an experiment is launched from the web interface, the web wrapper (*melcor-wrapper-web* in Figure 7) is required. The main wrapper (*melcor-wrapper*) is always required and is described in Section 3.2.1. *analytic1_editrules.cor* is a BRF which defines branching behavior (see Section 3.2.2). Only one BRF is submitted per experiment. *analytic1.cor.inp* is a TSIF which will form valid simulator input

when ADAPT variables are replaced with branching values (see Section 3.2.3). *analytic1.rst* is a simulator restart file which allows continuation from a saved state with new input going forward. The user will upload one TSIF and restart file for each simulator in the package. *melcor-checkpoint* is a checkpoint file which defines the command for ADAPT to use to pause the simulator. Generally, this involves creating a file whose presence the simulator recognizes as a stopping condition. Checkpointing allows an experiment to be paused in order to let a higher priority experiment run or accommodate system maintenance. Finally, *melcor* is a simulator executable which will be used to run branches[9].

After naming the files, the user proceeds by clicking ***Submit***. At this point the web wrapper and main wrapper files are uploaded. The next page allows the user to upload the files marked as `provided by installer` using the ***Choose File*** (Chrome) or ***Browse*** (Firefox) options as shown in Figure 8. Once files are defined their names are printed as shown in Figure 9. Note that the files uploaded in Figure 9 will be renamed to the names given in Figure 7.

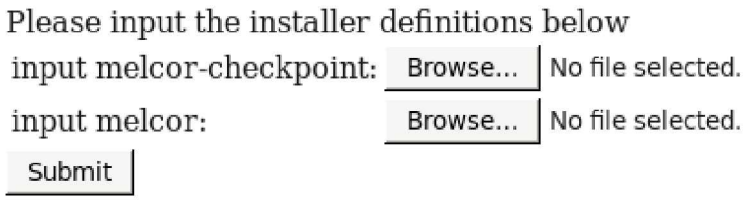

Figure 8: ADAPT Add Simulator: Locate Files to Upload

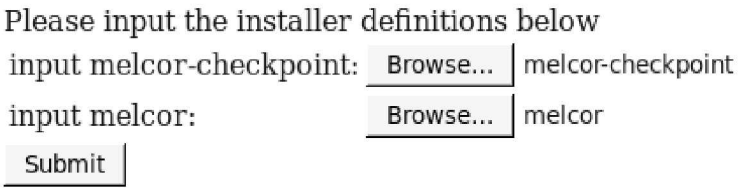

Figure 9: ADAPT Add Simulator: Located Files to Upload

Finally, clicking ***Submit*** will upload the remaining files and add the simulator to the SQL database. Figure 10 shows the message that is displayed after submission completes.

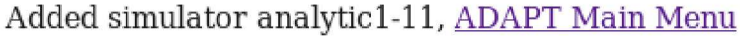

Figure 10: ADAPT Add Simulator: Submission Complete

[9]It is not strictly required to upload any of the files that appear under `Add inputs for simulator` in Figure 7. Files may be defined in the main wrapper and placed on a common filesystem manually. This method may be more practical if a large number of files are required.

2.3.2 Launching an Experiment

The ***Launch a new experiment using an existing simulator package*** option of the web interface (see Figure 5) allows the user to create a new ADAPT experiment (DET). The user chooses an installed simulator package (see Section 2.3.1) from the drop down menu. The user then names the experiment as desired and proceeds by clicking ***Submit*** as shown in Figure 11.

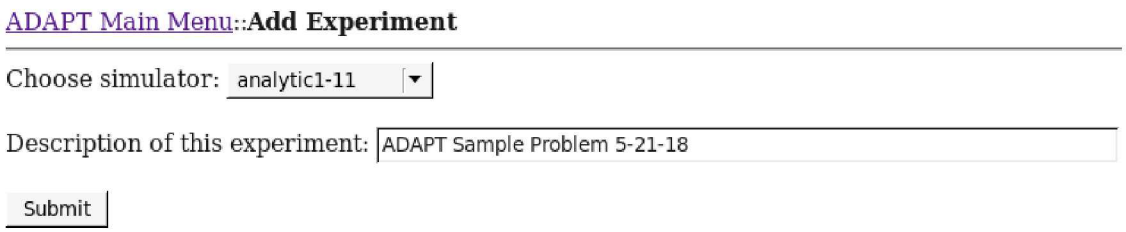

Figure 11: ADAPT Add Experiment: Choose Simulator Package and Name Experiment

Next, the user adds files for the experiment according to those marked as `provided by user` when creating the simulator package (see Figure 7). The interface provides the input file labels as specified when the simulator package was created as seen in Figure 12.

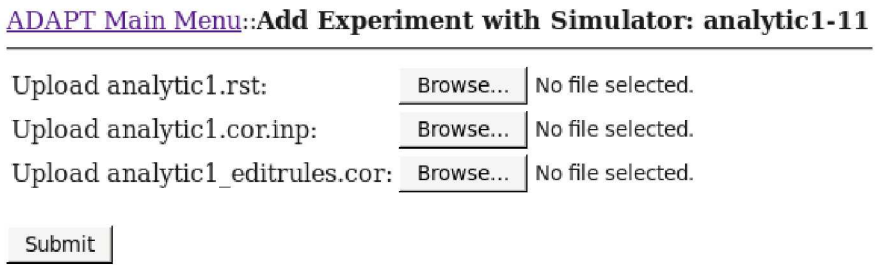

Figure 12: ADAPT Add Experiment: Locate Files to Upload

Using the ***Choose File*** (Chrome) or ***Browse*** (Firefox) buttons, the user finds the appropriate files on the local file system to be uploaded to the database. Figure 13 shows an experiment with all required files defined. The user then clicks ***Submit***, and the experiment is sent to the database queue to be run when resources are available. A successful submission yields a page similar to Figure 14. At this point the experiment has been added to the ADAPT database. The first branch of the new experiment will start according to its priority and position in the ADAPT queue. If an error such as a missing file occurs in submission, ADAPT will present the user with an appropriate error message. The user may correct the submission and then retry.

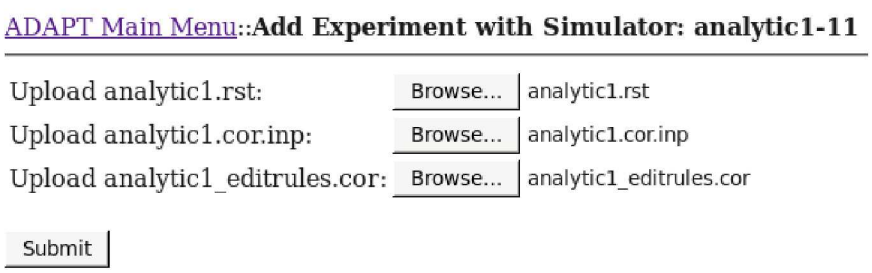

Figure 13: ADAPT Add Experiment: Located Files to Upload

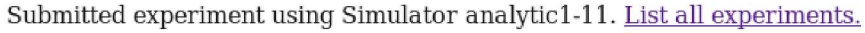

Figure 14: ADAPT Add Experiment: Submission Complete

2.3.3 Checking Progress

The ADAPT web interface produces a list of currently running and finished experiments under the `List all experiments` option in Figure 5. After being selected, a list of experiments run by the user appears as seen in Figure 15. Experiments are assigned an `Experiment#` based on their order of submission. Accompanying the `Experiment#` are columns for the `Name` given in the web wrapper and the `Description` given when submitting the experiment (see Section 2.3.2). The `State` column lists the status of each experiment such as `Active`, `Checkpointed`, or `Finished`. The number of `Total`, `Completed`, and currently `Running` branches are listed to provide the user with a brief snapshot of the progress of each experiment. Clicking `Runtime` leads the user to a page that displays the serial and parallel computational time that a simulation has required and the number of processing slots used. The next column displays `Restart`, `Checkpoint`, or nothing depending on the state of the experiment. If the state is `Active`, the experiment may be paused by clicking `Checkpoint`. If the state is already `Checkpointed`, clicking `Restart` will resume it. If the state is `Finished`, there is no action to be taken. The final column allows the user to `Delete` the experiment from the database.

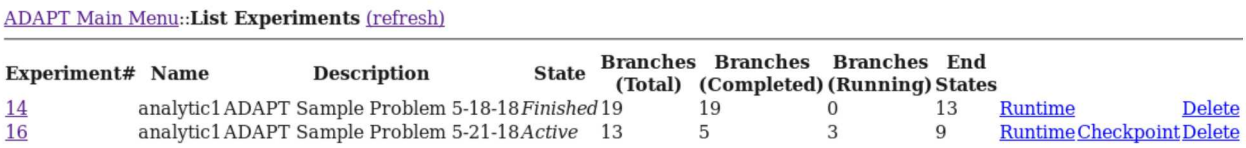

Figure 15: ADAPT List all experiments: all experiments listed

Selecting the ***Experiment#*** link for a certain experiment leads the user to a DET graphic as seen in Figure 16. The graphic shows the initial branch and all that follow as determined by the BRF. The DET is color-coded to signify if a branch has already run to completion (green), is currently running (cyan), is in the queue to run (yellow), or is checkpointed (blue). Examining the first branch, its unique branch number in the ADAPT database is `522`. Its probability is `1` and it is the `root` branch of the experiment. All other branches in the experiment will have a sub-branch

number such as `sb 1` to indicate which child branch of the branching condition is represented (see `BranchProbability` in Listing 14)[10]. The simulator that was run is `melcor`. The branch ended in a branching condition and its name is `Aleatory valve opening`. The branch stopped at a simulation time of `0.01` seconds and the branch took `17` seconds of wall clock time to run.

[10]Note that the sub-branch number is relative to the parent branch and is not unique across the experiment. The sub-branch number corresponds to the number (`1` or `2`) seen in the second column of lines 15-22 in Listing 14.

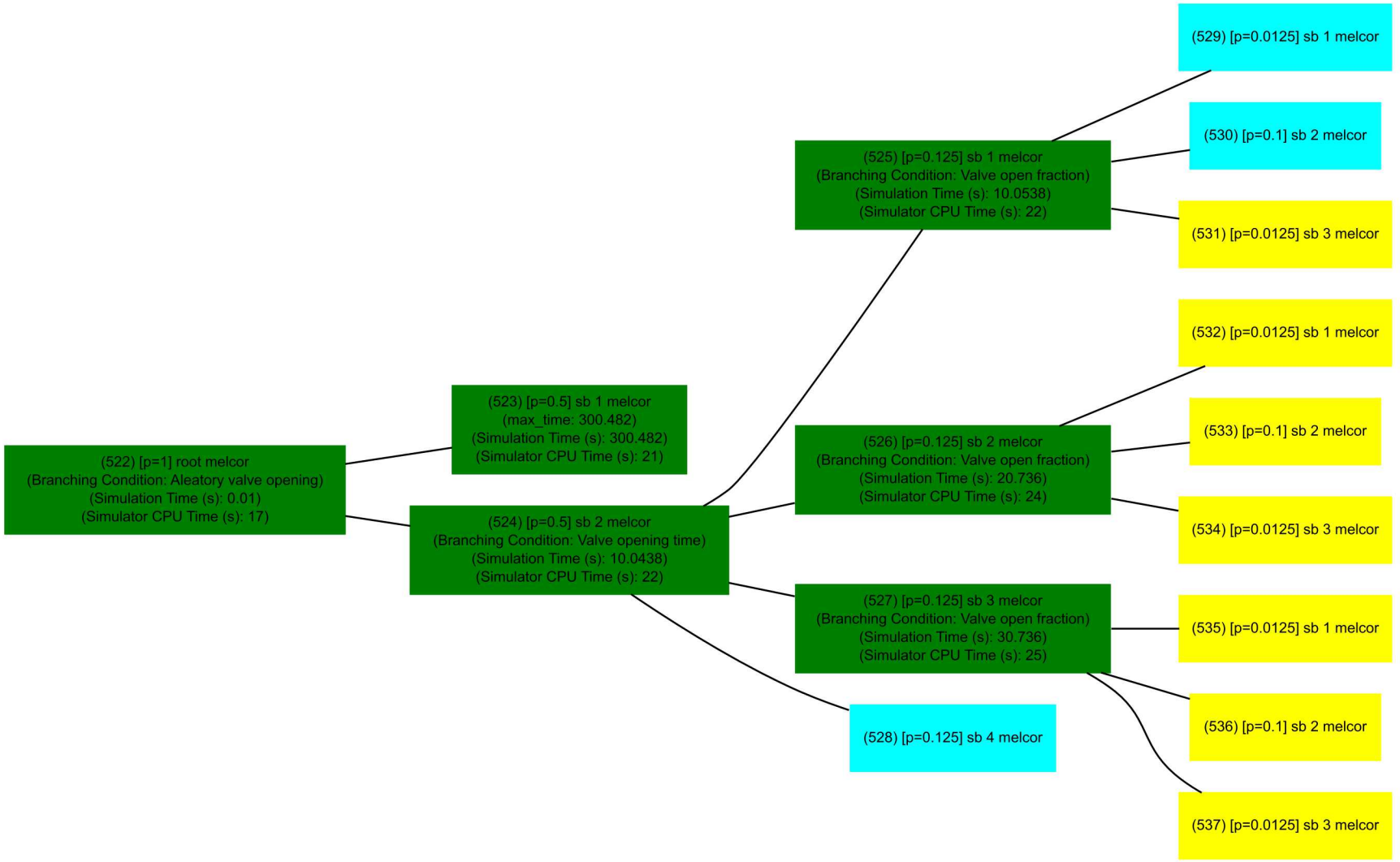

Figure 16: ADAPT Sample Problem DET Visualization

3 Input Creation

The process of input creation for ADAPT is described in this chapter. A sample problem is introduced in Section 3.1 based on a standard MELCOR assessment problem from Reference [39]. Section 3.2 describes the input files necessary for the ADAPT sample problem. The full set of inputs are given in Appendices C, D, and E[11]. The remainder of this chapter gives examples of how ADAPT cases are created and run.

3.1 ADAPT Sample Problem

Analytic problem 1 from Reference [39][12] was adapted to be an ADAPT sample problem. The problem was approved for release and appears in its entirety in Appendix E. Its initial conditions are tabulated in Table 3. In this problem, two control volumes (`CV001` and `CV002`) are initially separated by a closed valve across a flow path (`FL001`) as seen in Figure 17. One volume (`CV001`) is initially full of saturated water at high pressure and the other (`CV002`) contains low pressure steam. The valve opening is explored to produce an ADAPT sample problem. First, branching occurs on whether the valve will be opened. Next, if the valve will open, the valve opening time is determined. After branching for time, the valve opening fraction is determined and the valve is finally opened. The branching conditions for the sample problem are given in Table 4. The value presented in Table 4 is used in the simulator input, while the probability is tracked in the ADAPT database. The pressure of each control volume over time will be explored to demonstrate the output processing capabilities of ADAPT (see Section 4).

Table 3: ADAPT Sample Problem Initial Conditions [39]

Parameter	`CV001`	`CV002`
Pressure (MPa)	7.999	0.01
Temperature (K)	568.23	568.23
Water Mass (kg)	72240	0.0
Steam Mass (kg)	0.0	152.57
Void Fraction	0.0	1.0

[11] Another full set of ADAPT input files, representing an SFR accident using SAS4A/SASSYS-1, appear in Reference [16] with an analysis in Reference [30].

[12] An older version of the MELCOR analytic assessment problem input appears in Reference [40].

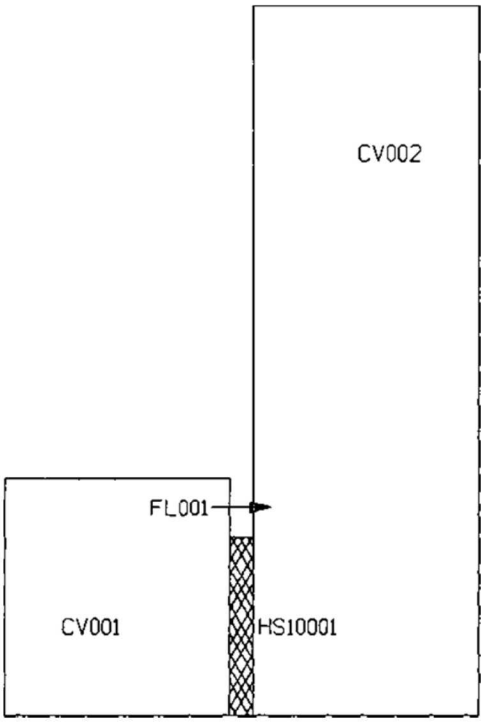

Figure 17: ADAPT Sample Problem Nodalization [39]

Table 4: ADAPT Sample Problem Branching Conditions

Branching Condition	Value	Probability
Valve Opens (Aleatory)	True	0.5
	False	0.5
Valve Open Time	10s	0.25
	20s	0.25
	30s	0.25
	40s	0.25
Valve Open Fraction	0.01	0.1
	0.5	0.8
	0.99	0.1

3.2 Required Input Files

This section describes the requirements for ADAPT input files and gives examples for the sample problem. Section 3.2.1 describes the wrapper file, Section 3.2.2 describes the BRF, and Section 3.2.3 describes the TSIF. When submitting an experiment through the web interface, ADAPT requires the following files:

- Web Wrapper: submits the experiment to the ADAPT database through the web interface
- Wrapper: defines the link between ADAPT and the simulator(s)
- Checkpoint File: defines how ADAPT can pause the simulator
- Branching Rules File (BRF): defines the actions to be taken when a branching condition is reached
- Simulator Restart File: contains the state of the system at the start of the case
- Template Simulator Input File (TSIF): contains information necessary to run the simulator with new values after branching
- Simulator Executable: runs the system model from the previously saved state with new input values

The Web Wrapper, Wrapper, Checkpoint File, and Simulator Executable(s) are submitted as part of a simulator package before any experiment may launch. The BRF, Simulator Restart File(s), and TSIF(s) are specific to each experiment.

3.2.1 Wrapper File

The wrapper file links either a simulator or a set of simulators [38] to ADAPT and must be created once for each unique set of simulators and post-processing activities. It may be reused for different DETs as long as the simulator has consistent input requirements, simulator output formats, and desired pre- and post-processing actions. The basic required actions of the wrapper file are given as Figure 18. If any of these required actions are not met, the experiment will not progress.

First, the appropriate input files are gathered and the simulator is started. There are three possible outcomes:

- The simulation may have reached its maximum time which will be detected by the wrapper and will result in no new branches.
- The simulator may fail which also results in no new branches.
- A branching condition may be indicated in the simulator output.

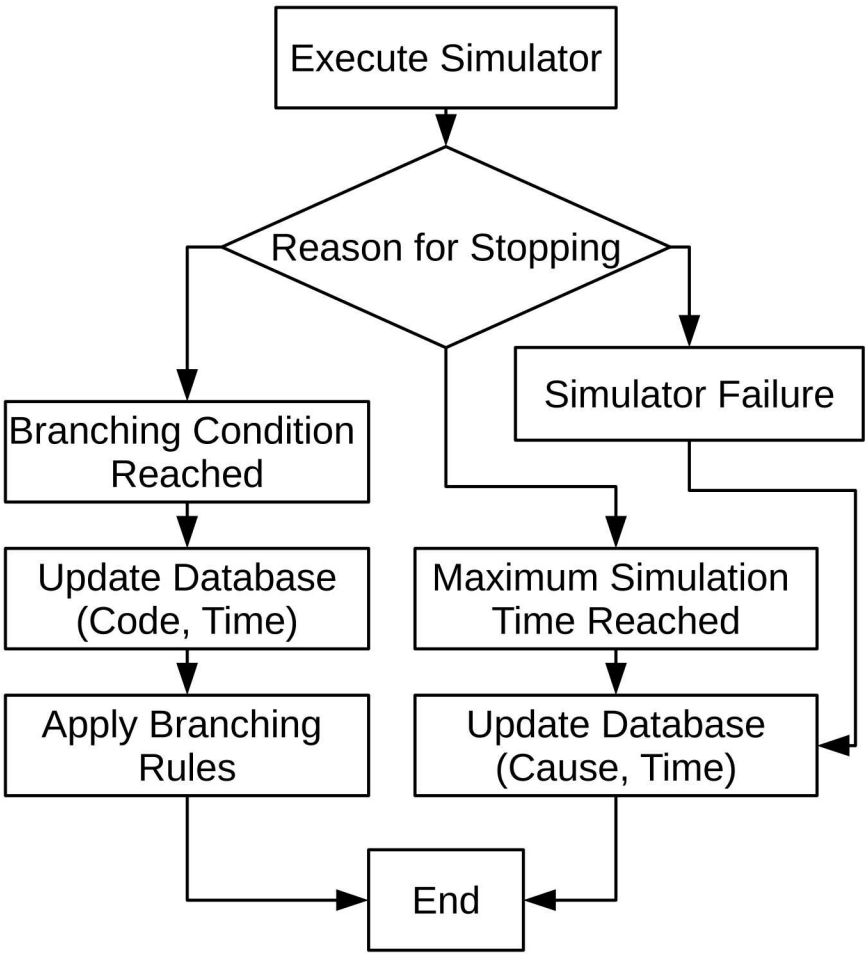

Figure 18: Required ADAPT Wrapper Actions

Failure of the simulator is detected by the exit status of the simulator or by an examination of the outputs. The results from the simulator are read through specially-created message files and the main simulator output files. If a branching condition is reached, the branching rules file is read to determine what variables need to change for sub-branches produced. The variable values and probabilities of the new (sub-)branches are submitted to the SQL database to join the queue. In addition, simulator restart and other output files are marked to be copied to the run directory of each new (sub-)branch. The primary changes to the wrapper for a new simulator are the execution syntax and the exact location in the message file of the reason and time for stoppage.

The first major operation in the wrapper script is to prepare the root branch. The portion of the script that performs this task is provided as Listing 10. The wrapper appears in its entirety in Appendix C. This piece of code is executed only if ***adapt-server*** has passed the `NCENGINE_ROOT` environment variable with value `1` and the branch is not being restarted from a checkpoint (line 1 in Listing 10). The restart file, TSIF, and BRF are copied from a central location to the directory of the first branch (lines 3-5). The branching rules are processed using the `INIT` values for all variables (lines 7-14) as described in Section 3.2.2.

Listing 10: ADAPT Root Branch Handling

```
1  if (str(os.getenv('NCENGINE_ROOT')) == '1') and (str(os.getenv
       ('NCENGINE_RESUMING_CHECKPOINT')) == '0'):
2      print('initializing edit rules for root job at %s' % (time
           .asctime()))
3      shutil.copy2(os.path.join(MELCOR_ROOT, RST), this_dir)
4      shutil.copy2(os.path.join(MELCOR_ROOT, SIM1TEMPLATE),
           this_dir)
5      shutil.copy2(os.path.join(MELCOR_ROOT, EDITRULES),
           this_dir)
6
7      editrule_cmd = 'adapt-editrule-apply --init %s %s 0 melcor
           %s' % (EDITRULES, BRANCHESF, OTHERTEMP)
8      print(editrule_cmd)
9      f = subprocess.Popen(editrule_cmd, shell=True, stdout=
           subprocess.PIPE, stderr=subprocess.STDOUT)
10     f_out = f.stdout.readlines()
11
12     # No loop here, because there is only one branch for the
           initial simulator for the root branch.
13     branchesf_content = open(BRANCHESF, 'r').readlines()[0]
14     shutil.copy2(branchesf_content.split()[0], os.path.join(
           this_dir, SIM1INPUT))
15     os.remove(branchesf_content.split()[0])
```

Listing 11 shows the logic used to start the simulator and terminate the job early if required by the BRF (see line 23 in Listing 14). Lines 1-3 of Listing 11 give the syntax of running the MELCOR simulator under ADAPT. Other simulators may require significantly different syntax. Note that line 3 uses the python ***subprocess.Popen*** method which starts a process and does not wait for a return signal before allowing subsequent code to run. This behavior is important to allowing early termination of jobs in ADAPT. If the `TERMINATE_EARLY` variable is passed by ***adapt-server*** with a value greater than `0`, the job will run for a number of seconds indicated by the value and then terminate (lines 4-10).

Listing 11: ADAPT Wrapper Simulator Execution and Early Termination

```
1  if THIS_EXECUTABLE == 'melcor':
2      sim1_exec_command = 'echo E | nice %s %s' % (os.path.join(
           MELCOR_ROOT, SIM1EXE), SIM1INPUT)
3      f = subprocess.Popen(sim1_exec_command, shell=True, stdout
           =subprocess.PIPE, stderr=subprocess.STDOUT)
4  if (os.getenv('NCENGINE_TERMINATE_EARLY') is not None) and (
       float(os.getenv('NCENGINE_TERMINATE_EARLY')) > 0.0):
5      time.sleep(float(os.getenv('NCENGINE_TERMINATE_EARLY')))
6      open('term-early', 'a').close()
7      open('analytic1.stp', 'a').close()
8  if os.path.isfile('term-early'):
9      f = subprocess.call(('adapt-job-record jobid %s term_early
           1' % (jobid)), shell=True)
10     sys.exit(0)
```

If the job did not terminate early, the exit status of the simulator is read by the wrapper. If an error is indicated, the job is marked as a simulator failure and no new branches are created. If the simulator closed successfully, the branching code (if any) is read from the message file and the simulation time is determined from the main output file. Listing 12 shows the portion of the wrapper that determines whether a branching condition was reached (see definition of `mystopping_code` in lines 3-6 of Listing 12) and what simulation time was reached (see definition of `sim_elapsed` in lines 7-10). Reading the stopping time and branching condition are the most simulator-dependent tasks of the wrapper as output formatting patterns differ by programming language and developer.

If a branching condition is indicated, the BRF is used as shown in Listing 13. While most outputs of ***adapt-editrule-apply*** are ignored in preparing the root branch (see Listing 10), all are used in creating child branches (see line 9 in Listing 13). The information is provided as a list of new branches each with:

- Simulator input file location (line 18 in Listing 13)
- ADAPT variable states (line 15 in Listing 13)
- Branching code of parent (line 23 in Listing 13)

- Probability (line 21 in Listing 13)
- Early termination status (line 12 in Listing 13)
- Simulator to run (line 13 in Listing 13)

Listing 12: ADAPT Wrapper Determination of Branching Condition and Simulation Time

```
1  melcor_message_file = 'analytic1.mes'
2  melcor_message_file_contents = open(melcor_message_file, 'r').
       readlines()
3  for line in reversed(melcor_message_file_contents):
4      if stop_word in line:
5          mystopping_code = line.split(stop_word)[1].split()[0]
6          break
7  for line in reversed(melcor_message_file_contents):
8      if line.startswith(' TIME='):
9          sim_elapsed = float(line.split(' TIME=')[1].split()
               [0])
10         break
11 for line in reversed(melcor_message_file_contents):
12     if line.startswith(' Normal termination  TIME='):
13         normal_term = float(line.split(' Normal termination
               TIME=')[1].split()[0])
14         sim_elapsed = normal_term
15         break
```

Characteristics including early termination status (`terminate_early`), the executable to be run (`newsim`), the probability (`probability`), and the branch number (`branchnum`) are submitted to the database for each new branch (see definition and execution of ***adapt-submit*** in lines 8-26 of Listing 13)[13]. The database is updated with every file the new branches will require to run. Small files are uploaded in their entirety to the database (`-handoff`) while for large files the location is submitted (`-handoffref`). After finishing this operation, the wrapper closes with exit code 0 which signals ***adapt-server*** that the branch is complete and a slot on its computation node may be marked as available for queued jobs. If a non-zero exit code is returned, the job is marked as failed and will be retried when a host is available.

[13] An optional flag may be passed to ***adapt-submit*** to define a priority for each new branch. Adding `-priority int` will ensure that the branch has priority `int` and will run before any branches with a lower priority number. By default, new branches are all given priority `1` and will run in order of submission.

Listing 13: ADAPT Wrapper Determination of New Branches

```
1  editrule_cmd = 'adapt-editrule-apply %s %s %s %s %s' % (
       EDITRULES, BRANCHESF, str(sim_elapsed), THIS_EXECUTABLE,
       OTHERTEMP)
2  print(editrule_cmd)
3  f = subprocess.Popen(editrule_cmd, shell=True, stdout=
       subprocess.PIPE, stderr=subprocess.STDOUT)
4  f_out = f.stdout.readlines()
5  branchesf_content = open(BRANCHESF, 'r').readlines()[0]
6  NEWSIM = branchesf_content.split()[7]
7  branchesf_content = open(BRANCHESF, 'r').readlines()
8  for line in branchesf_content:
9      (config, newstate, branchnum, stopcode, branchhit,
           probability, terminate_early, newsim) = line.split()
10     submit_cmd = ''
11     submit_cmd += 'adapt-submit                    '
12     submit_cmd += '--terminate_early %s            ' % (
           terminate_early)
13     submit_cmd += '--executable      %s            ' % (newsim)
14     submit_cmd += '--handoff         %s %s         ' % (EDITRULES,
           EDITRULES)
15     submit_cmd += '--handoff    %s %s.state        ' % (newstate,
           EDITRULES)
16     submit_cmd += '--handoffref %s %s              ' % (RST, RST)
17     submit_cmd += '--handoffref %s %s              ' % (
           SIM1TEMPLATE, SIM1TEMPLATE)
18     submit_cmd += '--handoff    %s %s              ' % (config,
           SIM1INPUT)
19     if os.path.isfile('analytic1.mes'):
20         submit_cmd += '--handoffref analytic1.mes analytic1.
               mes '
21     submit_cmd += '--probability %s ' % (probability)
22     submit_cmd += 'analytic1                       '
23     submit_cmd += '"sb %s"   ' % (branchnum)
24     submit_cmd += '%s' % (THISSCRIPT)
25     f = subprocess.Popen(submit_cmd, shell=True, stdout=
           subprocess.PIPE, stderr=subprocess.STDOUT)
26     f_out = f.stdout.readlines()
27     os.remove(newstate)
28     os.remove(config)
```

3.2.2 Branching Rules File (BRF)

A truncated BRF for MELCOR is given as Listing 14 and appears in full in Appendix D. This file contains information on how ADAPT should proceed when a branching condition is detected by the wrapper. A TSIF (see Section 3.2.3) is included in the input at the start of the DET and is progressively modified for each branching condition. The TSIF is given in the field `InputFile` (line 1 in Listing 14). Within the TSIF, ADAPT variables are set off using separators as defined in `VarSeparator` (line 3 in Listing 14). The message file associated with the simulator is found in the `StoppingWord` field as is a rule for searching the file (line 2 in Listing 14). In Listing 14's case ADAPT will search for the word `LIZARDKING` as the 2nd word of the last line in the file *analytic1.mes*. If found, ADAPT will recognize that a branching condition was reached. For this set of branching rules, ADAPT only performs an action if the branching condition is `20004` (lines 15-22 in Listing 14).

Listing 14: Sample ADAPT Branching Rule File

```
1  InputFile 1 analytic1.cor.inp
2  StoppingWord 1 analytic1.mes LIZARDKING 2
3  VarSeparator 1 "{" "}"
4  SimulatorExecutable 1 melcor
5  InitialSimulator 1
6  INIT V99999 FALSE
7  INIT V10000 0.0
8  INIT V20001 0.0
9  INIT V20003 TRUE
10 INIT V20004 FALSE
11 BranchingSimulator 20004 1
12 BranchProbability  20004 1 0.5
13 BranchProbability  20004 2 0.5
14 BranchingConditionName 20004 Aleatory valve opening
15 20004 1 V20003 FALSE
16 20004 1 V20004 FALSE
17 20004 1 V20011 1.e20
18 20004 1 V20021 1.e20
19 20004 1 V99999 FALSE
20 20004 2 V20003 FALSE
21 20004 2 V20004 FALSE
22 20004 2 V99999 FALSE
23 TerminateEarly 60 20004 1
```

It is possible to name branching conditions in ADAPT by setting `BranchingConditionName` as seen in line 14 of Listing 14. Assigning names to branching conditions will affect the visualization of the DET (see Figure 19). If no name is given, the branching condition identifier (e.g. `20004`) is used in the visualization. It is possible to terminate a job early if its simulator results are of little interest. The input shown in line 23 of Listing 14 will end a job that follows the first (`1`) sub-branch of branching condition `20004` after `60` seconds (see Section 3.2.1).

In the case of Listing 14, a branch that ends by meeting condition 20004 will lead to two child branches (lines 12-13). The branches differ in the values of V20011 and V20021 which represent triggers for future branching conditions. In both cases the variable V20003 is changed from a value of TRUE to a value of FALSE (lines 15 and 20). For this specific input file, that change has the effect of disabling branching condition 20004 in the future and ensuring that it occurs at most once in any final state in the DET. Branching conditions that are allowed to occur multiple times in a DET commonly must be crafted with a refractory period in the simulator input file to avoid their occurring inadvertently in quick succession. This input is consistent with ADAPT input format changes introduced in Reference [38].

3.2.3 Template Simulator Input File (TSIF)

The TSIF used with ADAPT does not typically contain all of the input used in the simulation model. The TSIF in Appendix E does contain all input for the sample problem. The TSIF is only required to contain the input that is necessary to restart the simulation and that which may change with branching. Reducing the size of the TSIF may contribute to saving time and disk space over the course of a DET particularly if the full input is very large and many branches are created.

A portion of the TSIF is given as Listing 15. The text set off by "{" and "}" (e.g., line 4 of Listing 15) are ADAPT variables to be defined by the branching rules file (see Listing 14). When branching condition 20004 from Listing 14 occurs, the second branch replaces {V20003} with FALSE (line 20 in Listing 14), {V20004} with FALSE (line 21 in Listing 14), and {V99999} with FALSE (line 22 in Listing 14) to create a valid MELCOR input file for that branch. These changes take effect in lines 8, 10, and 12 of Listing 15, respectively. These changes have the effect of disabling branching condition 20004 for the future of the DET (V20003), changing the branching condition control function's value (V20004) from true to false immediately, and changing the MELCOR stop control function's value from true to false (V99999).

Listing 15: Portion of ADAPT Template Simulator Input File

```
1  Program MELCOR
2  CF_INPUT
3  cf_id 'FlowFrac' 10000 read
4  cf_sai 1.0 0.0 {V10000}
5  cf_id 'Stp-Alea-Vlv' 20001 read
6  cf_sai 1.0 0.0 {V20001}
7  cf_id 'Stp-Alea-Arm' 20003 1-read
8  cf_liv {V20003}
9  cf_id 'Stp-Alea-Go' 20004 1-and
10 cf_liv {V20004}
11 cf_id 'ADAPTCF' 99999 L-OR
12 cf_liv {V99999}
```

3.3 Using Multiple Simulators

The use of multiple simulators under ADAPT was first described in Reference [38]. This feature allows a single DET to address more phenomena or address phenomena in greater fidelity than would be feasible with a single simulator. Care must be taken to consider all intended transitions between simulators. For example, an analysis linking MELCOR (a severe accident analysis code) and MACCS (a release consequence analysis code) may allow transitions from MELCOR to MELCOR, from MELCOR to MACCS, and from MACCS to MACCS. Due to dependencies, it would not be expected for most analyses to transition from MACCS to MELCOR. On the other hand, an analysis that links MELCOR with a fluid dynamics code may transition between the codes multiple times depending on when uncertainties are encountered that may only be resolved by the fluid dynamics code.

When using multiple simulators, there is still only one BRF required for an experiment. Note the locations in Listing 14 where an input parameter requires a simulator number. For example, the TSIF (`InputFile`), the word to search for to indicate a branching condition (`StoppingWord`), and the ADAPT variable separators (`VarSeparator`) all vary by simulator. A name for each simulator is defined by `SimulatorExecutable` (see line 4 in Listing 14). This name is passed to the wrapper to identify which simulator should be run and how pre- and post-processing should proceed. The `InitialSimulator` input gives the number of the simulator to run for the first branch of the DET. The simulator to use for subsequent branches is specified using `BranchingSimulator`.

The user will likely upload a TSIF, a restart file, and an executable for each simulator. Logic will be required in the wrapper to use different syntax for the different simulators as seen in line 1 of Listing 11. There will likely be different methods required to retrieve the simulation time and branching condition (if one is met) after the simulator has finished. If a branching condition has been met, the outputs of the simulator just run may need to be translated in some way to be used as input to the next simulator to be run. This translation work is not described in this manual as it will vary significantly between different sets of simulators and for different transitions within the same set of simulators. Examples of the translation process are given in References [21] and [38]. The files to be passed to each child branch may differ depending on the transition (see Listing 13). At a minimum, the BRF (line 14 in Listing 13), the state of the ADAPT variables (line 15), the TSIF for each simulator (line 17), and the branch input file (line 18) should be passed to each child branch.

4 Output Processing

This section gives an overview of the output processing capabilities of ADAPT from the command line. Section 4.1 describes the process of collecting output data from an ADAPT experiment including defining the format of the output being gathered and describing how to use the functions to stitch the parent-child relationship of branches together. Section 4.2 describes how to plot the horsetails of a given experiment. Section 4.3 describes how to apply importance measures to an experiment to help identify important branching conditions. Most of the these capabilities may be accessed through the ADAPT web interface which has its own internal help content. These tools are applied to the sample problem (see Section 3.1) whose complete DET appears in Figure 19.

4.1 Gathering Outputs

The simulator outputs need to be extracted and sorted to allow for subsequent post-processing. Section 4.1.1 describes how to tell ADAPT the format of the data to be post-processed. Section 4.1.2 describes the set of functions needed to sort and define the parent-child relations of the various branches.

4.1.1 Output Formatting

The built-in ADAPT output processing functions currently expect time-dependent data. ADAPT expects the data for each variable to be in a separate file. The name of the data file for a particular variable should be consistent between all branches, i.e., it should not include the branch number or a time stamp. The requirements for file contents are as follows:

- There are no headers. The first line of the file is the first line of data. The variable should be indicated by the file name.
- Each line consists of a time string followed by a value string. ADAPT evaluates the strings as floating point numbers.
- Times and values may be separated by a space ( ), tab (\t), comma (,), colon (:), or semicolon (;).

An example of the expected content format is shown in Listing 16 representing the pressure of volume `CV001` of the sample problem (see Section 3.1). The Python ***float*** function is used to translate each time or value string to a floating point number.

Listing 16: ADAPT Plot File Format Example

```
1 30.736031  7999044.0
2 32.311981  7998665.0
3 34.102798  7998225.5
4 36.041069  7997743.5
```

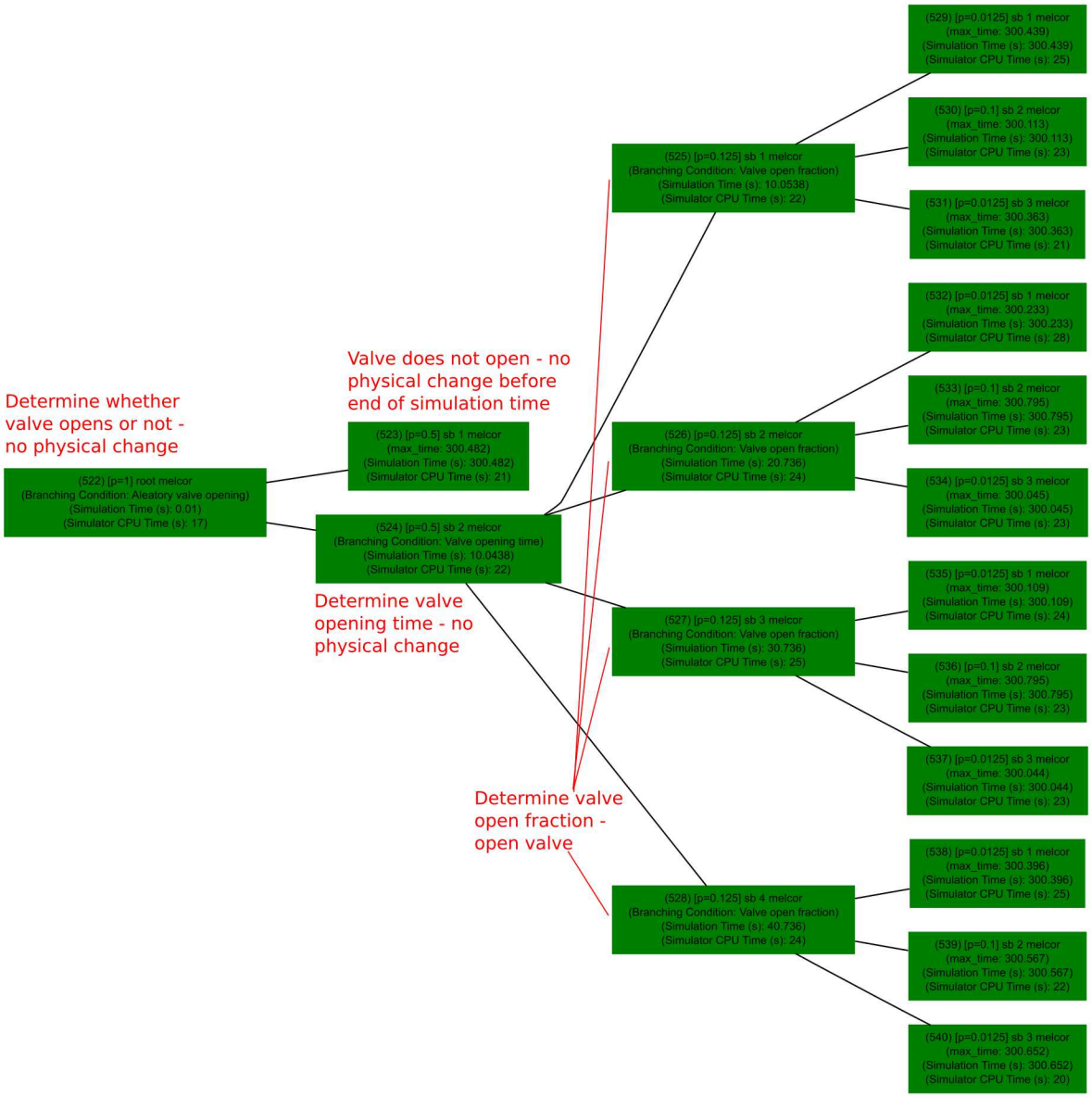

Figure 19: ADAPT Sample Problem DET Visualization

Green: finished branch, Cyan: running branch, Yellow: queued branch

4.1.2 Gathering Functions

ADAPT contains multiple functions that may be used to gather plot file information. Some of them may be run in parallel for significant time savings over serial operation. These functions are listed in Table 5 as a quick reference.

Table 5: ADAPT Data Gathering Functions

Function Name	Description
get_plot_data	Retrieves plot information for input hostname and absolute path
get_branch_plot_data	Retrieves plot information for input plot file name and branch number
get_heritage_plot_data	Retrieves concatenated plot information from root branch for input plot file name and branch number
get_exp_heritage_plot_data	Retrieves concatenated plot information for all sequences for input plot file name and experiment number
get_exp_branch_heritage	Retrieves branch heritage information for input experiment number

The ADAPT function ***get_plot_data*** is used to retrieve formatted information from a single plot file from a single directory. The host and absolute path must be known. Information is returned by ***get_plot_data*** as a list of lists if the file can be read and is in the expected format (see Listing 16). The outer list is a set of elements based on the lines of the file. The inner list is the time followed by the variable value. For example, the information returned from the data file represented by Listing 16 is:

```
[[30.736031, 7999044.0], [32.311981, 7998665.0], [34.102798, 7998225.5],
                         [36.041069, 7997743.5]]
```

If the file cannot be read or the contents are not in the expected format, ***get_plot_data*** returns `False`. A related function, ***get_branch_plot_data***, allows the user to specify a plot file and a branch number in the ADAPT database to find the host and directory for the specified branch. After finding the host and directory, ***get_branch_plot_data*** calls ***get_plot_data***.

The function ***get_heritage_plot_data*** concatenates information from the specified plot file for the specified branch as well as the string of branches leading back to the first branch of the experiment. First, the heritage of the branch is determined using the function ***get_branch_heritage*** (see Appendix B). Next, ***get_heritage_plot_data*** calls ***get_branch_plot_data*** for each branch in the sequence. Branches with no valid information for the specified plot file are skipped. The entries are concatenated and sorted by time. Duplicate entries may occur depending on the plotting rules of the simulator. These entries are identified by matching both time and value and are eliminated. As with the previous functions, ***get_heritage_plot_data*** will return `False` if no valid information has been gathered.

The function ***get_exp_heritage_plot_data*** concatenates information from all sequences for the specified plot file and experiment number. This process may become computationally expensive and steps have been taken to reduce the time required to run. The end branches of the experiment are identified using ***get_exp_branch_heritage*** (see Appendix B)[14]. A previously-gathered heritage data file may be used if it exists and `use_old_data` was specified as `True` at the time of calling ***get_exp_heritage_plot_data***. Invoking this option may save significant time for a large experiment if many variables are to be gathered.

If the information for the specified plot file has been gathered before, it may be re-used. This feature may be particularly useful when the user desires a snapshot of a DET in progress. Meta-information is written by ***get_exp_heritage_plot_data*** regarding the plot file when it is run. This information includes the names of branch (job) directories, the size of each plot file, and the modification time of each plot file. If either no previously-gathered information exists or the user has not specified to use it, the information is gathered for each end branch. Gathering is accomplished using ***get_plot_data*** which may be run in serial or parallel depending on whether `use_processors` was specified with a value of `2` or greater[15].

The meta-information and plot information produced by ***get_exp_heritage_plot_data*** are stored in the *plot_cache* sub-directory within the *server_adaptemp* directory (see Listing 2). If gathered for many variables and experiments, this cached information may tax the storage capabilities of the host machine. All information in this directory may be safely deleted without any loss of underlying data from the experiment.

4.2 Plotting Outputs

There is an executable script (***adapt-plot-horsetails***) within ADAPT that produces plots of time-dependent simulator outputs. This script uses the information retrieved by the tools from Section 4.1.2 to show the range of a single variable across all sequences in a DET. The required inputs for ***adapt-plot-horsetails*** are an experiment number, a plot file, and a y-axis label. The x-axis label defaults to "`Time (s)`". An example of a plot output is shown in Figure 20 which plots the pressure of `CV001` in the ADAPT sample problem (see Section 3.1). The top of the plot includes information such as the plot file used, the time the plot was produced, and the input options used. In the case of Figure 20, each sequence may be easily matched to the branching conditions from Table 4.

There are a number of optional inputs to ***adapt-plot-horsetails*** which are listed in Table 6. The calculation of the median and mean values across all sequences at each time step may be computationally costly. The cost may be somewhat reduced by using multiple processors and by binning time steps. By default, 100 bins are used to divide the time span linearly. Either of these parameters may be changed through optional input in Table 6. The plot file information is sorted by assigning each time step of each sequence to a bin. For the purposes of selecting a 50th percentile

[14]End branches are those with no identified child branches. A branch is an end branch while running but may no longer be if it yields one or more child branches.

[15]This operation is not typically processor-bound on modern systems and the value of `use_processors` that minimizes time required may exceed the number of physical cores.

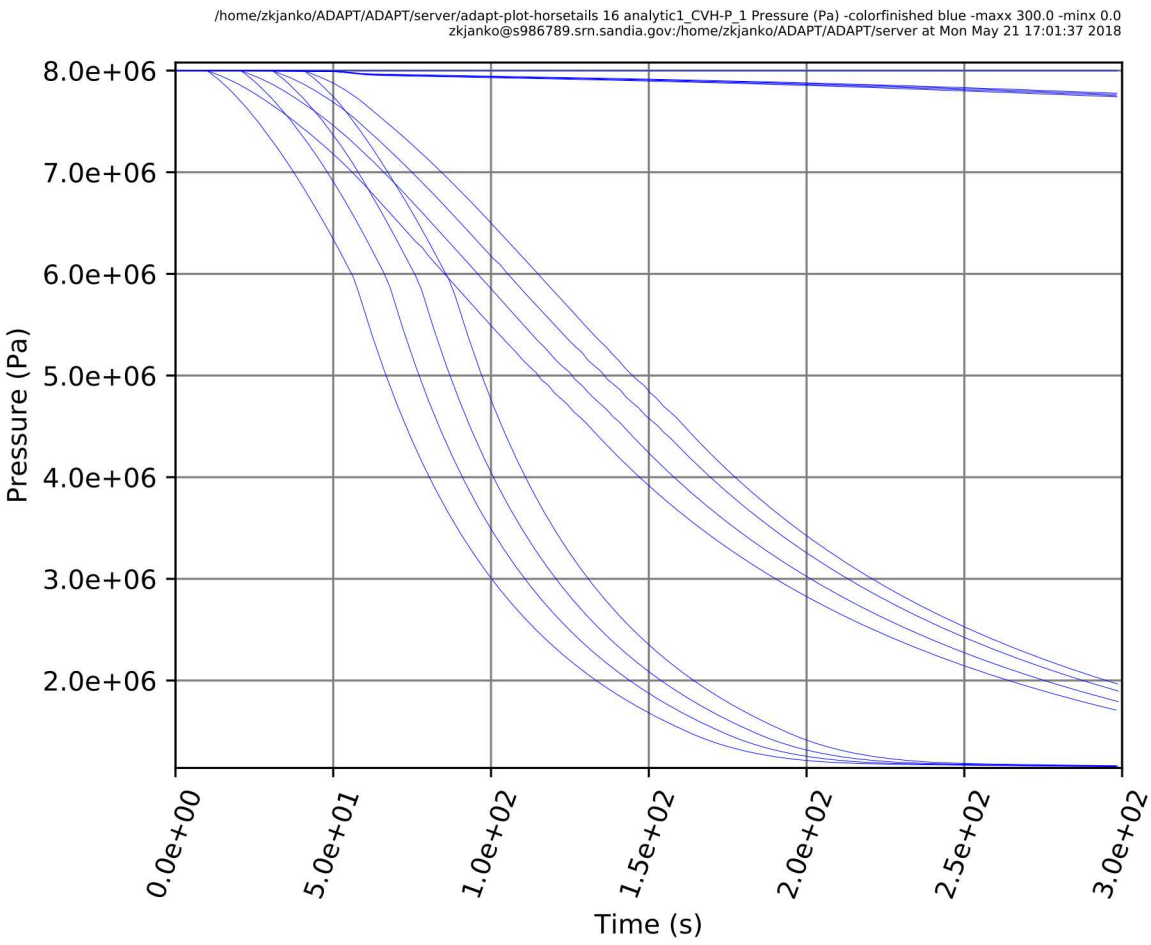

Figure 20: Sample *adapt-plot-horsetails* Plot

value, if multiple time steps from a sequence are assigned to the same bin they are collapsed into a single value with the average time and value of the included steps. When calculating the median, the plot values in each bin are sorted and sequence probabilities are used to find the 50th percentile sequence. All points from the 50th percentile sequence for each bin are returned for plotting by ***adapt-plot-horsetails***. When calculating the mean, the weighted average plot value (and time midpoint) of each bin are returned for plotting.

Table 6: Optional Input for *adapt-plot-horsetails*

Optional Input	Python Data Type	Description
`-maxy` *x*	float	Set the upper y-axis window limit to *x*.
`-miny` *x*	float	Set the lower y-axis window limit to *x*.
`-maxx` *x*	float	Set the upper x-axis window limit to *x*.
`-minx` *x*	float	Set the lower x-axis window limit to *x*.
`-colorfinished` *x*	str	Plot finished sequences in color *x*.
`-colorunfinished` *x*	str	Plot unfinished sequences in color *x*.
`-useolddata` *x*	bool	Use old data if it is available.
`-useprocessors` *x*	int	Use *x* processors if new data must be pulled.
`-median` *x*	str	Plot the median value at each time step in color *x*.
`-mean` *x*	str	Plot the mean value at each time step in color *x*.
`-bins` *x*	int	Use *x* bins for determination of median and mean.
`-bintype` *x*	str	Divide time steps for binning either linear or log-10. Options are “lin” and “log”.
`-legend` *x*	bool	Do or do not print a legend for mean/median.
`-maxtimegap` *x*	float	Do not print subsequent points in a sequence if there is a time gap greater than *x*.

A plot is shown in Figure 21 with several optional inputs specified. Median values are plotted in red. The default 100 linearly-divided bins are used. The mean appears as a heavy dashed blue line.

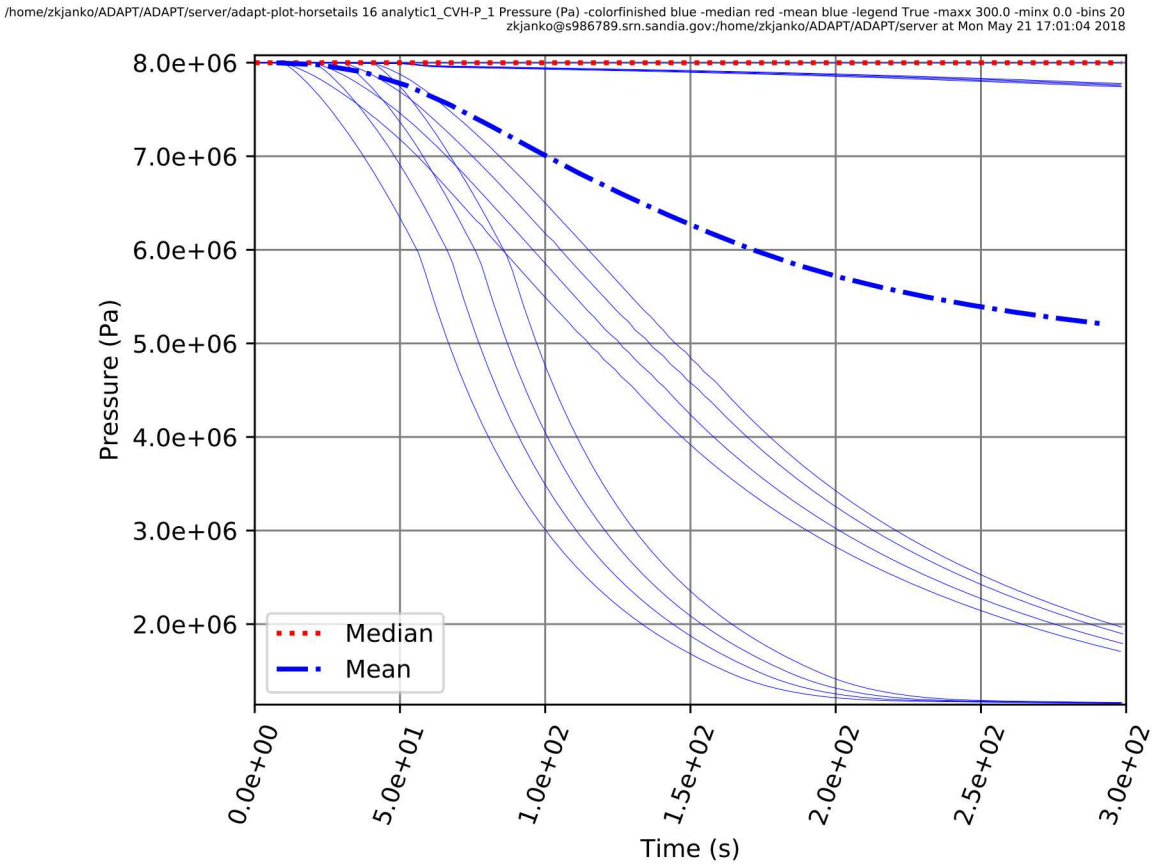

Figure 21: Sample *adapt-plot-horsetails* Plot, Optional Inputs

4.3 Calculating Importance Measures

The ADAPT function ***dynamic_importance_measure_calculation*** allows the comparison of values of a consequence measure across different sets of sequences within an experiment. Three measures called Dynamic Importance (DYI) are currently included and are listed in Table 7. DYI1 calculates the ratio of the expected values of the chosen consequence measure when an event occurs ($x = 1$) to when it does not occur ($x = 0$). An example of a DYI1 calculation may be "the ratio of expected pressure for valve opening versus no opening."

Table 7: DYI Dynamic Importance Measures

Importance Measure	Description
$DYI1 = \frac{R(x=1)}{R(x=0)}$	Consequence ratio of occurrence to non-occurrence
$DYI2(i) = \frac{R(x=1_i)}{R(x=0)}$	Consequence ratio of occurrence value $x = 1_i$ to non-occurrence
$DYI3(i) = \frac{R(x=1_i)}{R(x=1)}$	Consequence ratio of occurrence value $x = 1_i$ to all occurrence $x = 1$

DYI2 calculates the ratio of the expected values of the chosen consequence measure when an event occurs with some timing or severity i ($x = 1_i$) to when it does not occur. An example calculation may be "the ratio of expected pressure for valve opening at 10s versus no opening." Finally, DYI3 calculates the ratio of the expected values of the chosen consequence measure when an event occurs with some timing or severity i ($x = 1_i$) to all instances of occurrence ($x = 1$). An example calculation may be "the ratio of expected pressure for valve opening at 20s versus all opening times." DYI3 applies to any continuous input parameter and does not necessarily need to represent an "event."

The ***dynamic_importance_measure_calculation*** function requires multiple inputs. A sample set of input is given in Table 8 corresponding to the format shown in Listing 17. The inputs `scaled`, `use_old_data`, and `use_processors` are optional. The analysis begins by collecting all sequences from the input experiment where the input branching condition occurred using the helper function ***find_var_relevant_branches_and_end_states*** (see Appendix B). This step returns both the values of the input ADAPT variable and the probability for each sequence. Sequences are sorted by the final value of the input ADAPT variable[16]. This method allows the sorting of sequences according to whether an event occurred, did not occur, or occurred with some specific value of timing or severity.

Listing 17: DYI Input Format (see Table 8 for sample values)

```
dynamic_importance_measure_calculation(expid, bc_chosen,
    var_chosen, DYI_name, DYI_type, consequence_file,
    var_value_nonoccurrence, consequence_type)
```

[16]This does not currently recognize the situation where a branching event occurs multiple times. The last value of the ADAPT variable in the sequence will be used.

Table 8: DYI Input Example (see Listing 17 for format)

Input	Description	Sample Value
expid	Experiment number from database	`16`
bc_chosen	Branching condition from BRF	`'20004'`
var_chosen	ADAPT variable from BRF	`'V20011'`
DYI_name	Descriptive name for calculation	`'Valve Opening P1'`
DYI_type	DYI type (currently `1`, `2`, `3`)	`1`
consequence_file	Name of plot file to use for consequence	`'analytic1_CVH-P_1'`
var_value_nonoccurrence	Value of var_chosen when an event has not occurred	`'1.e20'`
consequence_type	Type of calculation (`'PEAK'`, `'VALLEY'`,`'LAST'`, `'MIDPOINT'`, `'ALL'`)	`'LAST'`
scaled	Whether or not to scale consequence to vary from 0 to 1 (default `False`)	`False`
use_old_data	Whether or not to use old data if available (default `False`)	`True`
use_processors	Number of processors to use in gathering data (default `1`)	`3`

Next, information from the input consequence measure file is loaded using ***get_heritage_plot_data*** for the relevant sequences (see Section 4.1.2). Depending on the input consequence type, ***dynamic_importance_measure_calculation*** will use the final value for each sequence (`'LAST'`), the peak value for each sequence (`'PEAK'`), the lowest value for each sequence (`'VALLEY'`), the time midpoint value of each sequence (`'MIDPOINT'`), or all values for each sequence (`'ALL'`). Because plotting time intervals may vary across sequences, if all values are to be used for DYI calculation, the information is binned. The sequence with the most time steps is used as a reference, and each time step of each other sequence is matched to one of the reference time steps.

The ADAPT variable values, sequence probabilities, and consequence values are used to calculate the expected value for each set of sequences (e.g., "event occurred") required according to the input DYI type. The sample event tree given in Figure 19 was subjected to a sample DYI1 calculation in Equation 1. Equation 1 compares the lowest values of the measure when the valve opens versus when it does not. An interpretation of the results of Equation 1 is that the expected lowest pressure when the valve opens is 0.21 times the expected lowest pressure when the valve does not open. Further analysis of this DET using DYIs appears in Appendix F.

$$DYI1 = \frac{R(x=1)}{R(x=0)} = \frac{\sum_{i=1}^{n_{x=1}} P_i \cdot Cs_i}{\sum_{i=1}^{n_{x=1}} P_i \cdot Cs_i} = \frac{\left(\frac{0.5 \cdot 1659817.9}{0.5}\right)}{\left(\frac{0.5 \cdot 7999044.0}{0.5}\right)} = \frac{1659817.9}{7999044.0} = 0.2075 \quad (1)$$

4.4 Trimming Trees

Just as uncontrolled growth may be detrimental to a forest, an overly large DET may be difficult to manage. Traditional event trees are commonly broken into sections of at most a few dozen end states for better readability. The scale of a DET has historically ranged from hundreds to millions of total branches (see Section 1.4.4). The potential scale of these DETs impedes the manual simultaneous examination of the entire tree. To increase the tractability of DET output processing, a tool is included in ADAPT that reduces a DET according to a user-input set of rules for easier manipulation by the analyst [37]. This tool is accessible from the ADAPT web interface (see Figure 5). First, a base experiment is chosen and a description of the reduction is given as in Figure 22.

In order to reduce the copied DET, the user must input decision rules. A rule associated with the sample problem from Section 3.1 is given in Table 9. The meaning of the valid `Parameter Operator` and `Time Operator` values are given in Table 10[17]. In this example, the user wishes to return only sequences where the time series value of `analytic1_CVH-P_2` (which represents the pressure of `CV002`) exceeds `50000.0` at a time before `150s`. The user inputs rules in the web interface as seen in Figure 23.

Figure 24 shows the full plot of `CV002` pressure for the ADAPT sample problem. The plot in Figure 25 has been reduced according to the rule in Table 9. It may be seen by comparing the figures that the reduction rule eliminated the sequences where the valve did not open and where it opened with a fraction of 0.01 (see Table 4). The visualization of the reduced DET is given as Figure 26 (compare to Figure 19). Further analysis of the results of this tree reduction may be seen in Appendix F.

Figure 22: Reduce Experiment: Choose Base and Description

[17] In practice, it is difficult to match an exact time or parameter value (operator value 3). Values are read into ADAPT as strings and converted to floating point numbers.

Rule 1 name: high CV002 pressure early

Rule 1 parameter (name of data file): analytic1_CVH-P_2

Rule 1 value: 50000.0

Rule 1 value operator: 1

Rule 1 time: 150.0

Rule 1 time operator: 2

Figure 23: Reduce Experiment: Set Rules

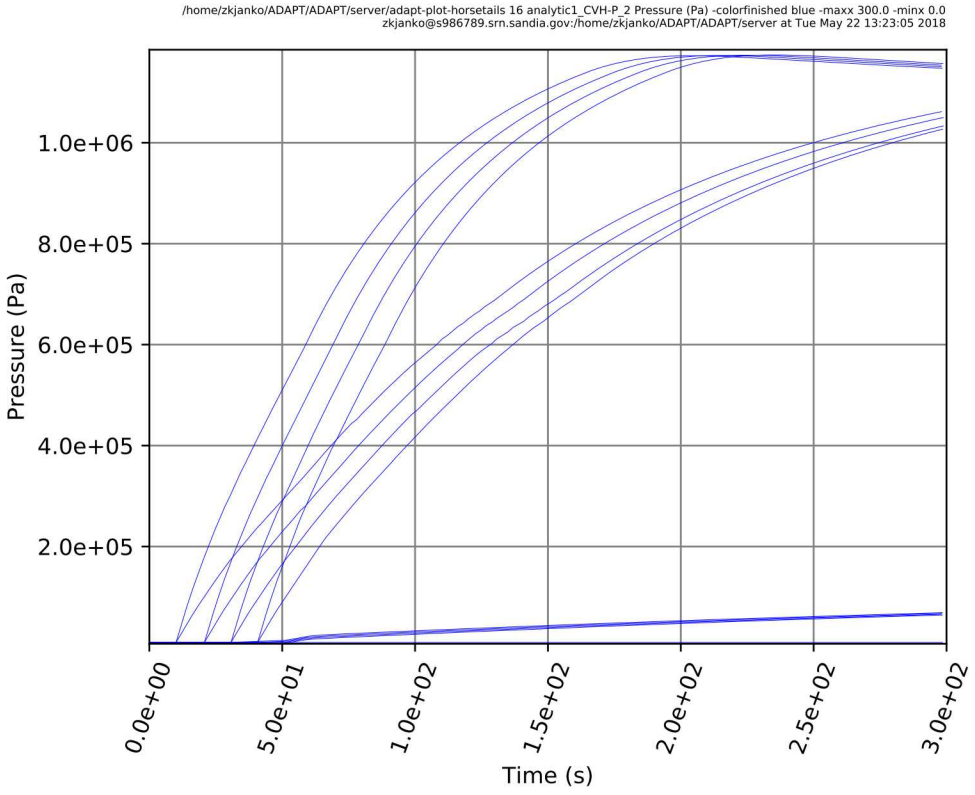

Figure 24: ADAPT Sample Problem CV002 Pressure Plot

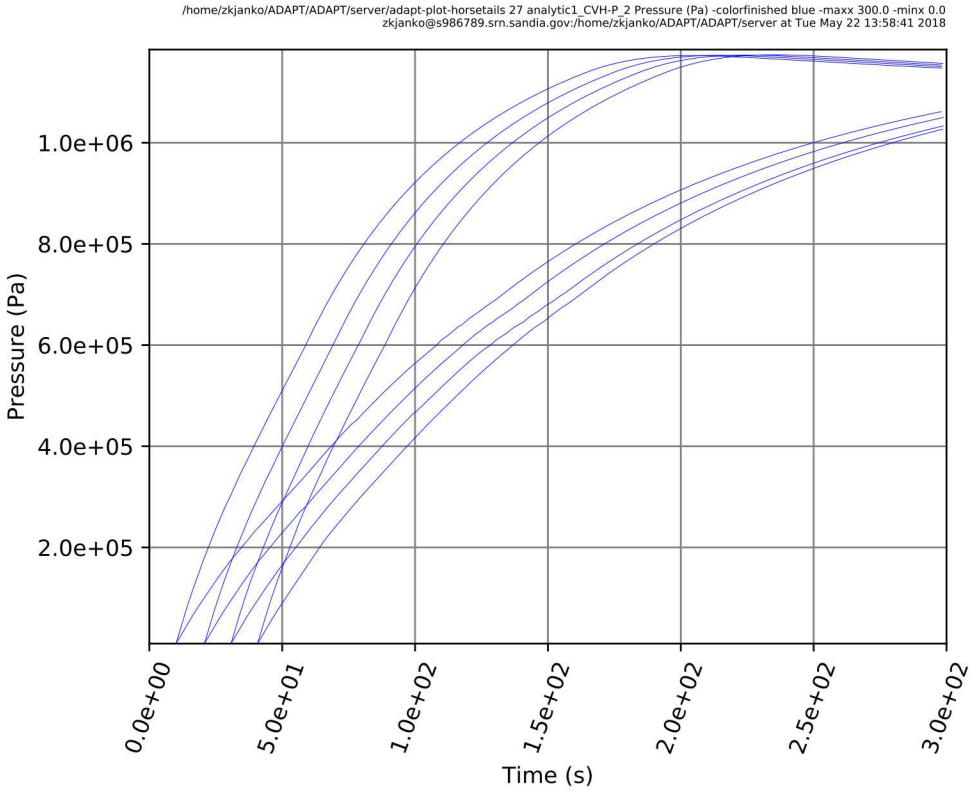

Figure 25: ADAPT Sample Problem `CV002` Pressure Plot, Reduced by Table 9 Rule

Table 9: Sample DET Reduction Rule

Input Field	Rule 1
Name	`High CV002 Pressure Early`
File	`analytic1_CVH-P_2`
Parameter Value	`50000.0`
Parameter Operator	`1`
Time Value	`150 seconds`
Time Operator	`2`

Table 10: Parameter and Time Operator Values for Reduction Rules. Refers to File variable in Table 9

Operator	Intent
`1`	File value greater than specified value
`2`	File value less than specified value
`3`	File value equal to specified value

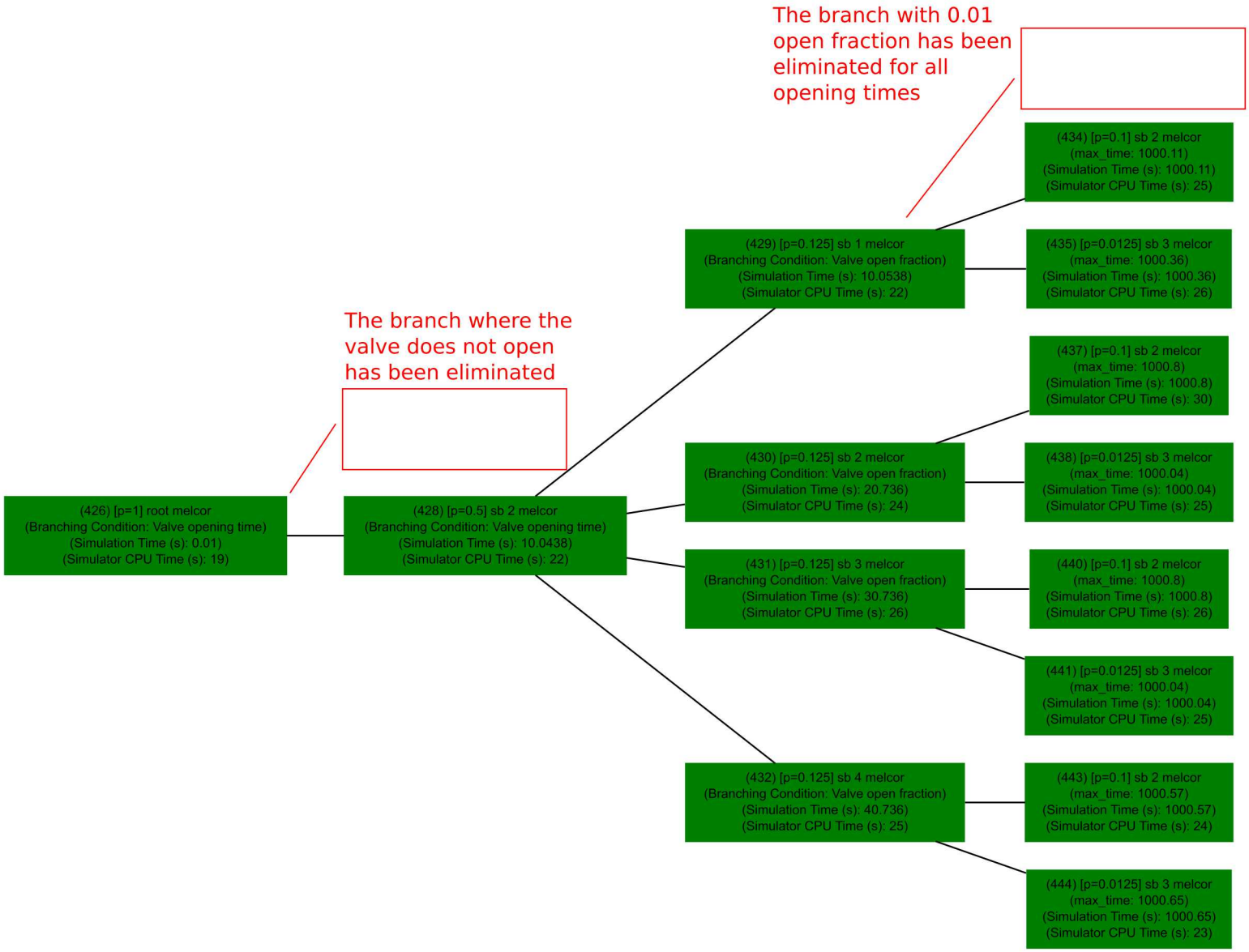

Figure 26: ADAPT Sample Problem DET Visualization, Reduced by Table 9 Rule

Green: finished branch, Cyan: running branch, Yellow: queued branch

5 Conclusion

This manual is to serve as a reference and tutorial for ADAPT as well as a roadmap to the future of ADAPT. Some brief concluding remarks are given in Section 5.1 and future plans are outlined in Section 5.2.

5.1 Concluding Remarks

This manual has given a basic introduction to dynamic PRA and ADAPT. ADAPT can be valuable in discovering insights in complex system transients. Dynamic PRA in general is expected to take a larger role in the safety analysis of advanced reactors. When compared to traditional light water reactors, advanced reactor designs tend to credit more natural phenomena (e.g., natural circulation) in lieu of active safety systems. This is an area where dynamic PRA is recognized as having advantages over the traditional ETFT method (see Section 1.3).

ADAPT links different simulators in a variety of schemes based on the problem being analyzed. This manual has described the input and outputs of a simple analysis in order to reflect the necessary elements. Requests for clarification or proposals for collaboration may be directed to `ADAPT@sandia.gov`.

5.2 Future Work

This manual will be updated periodically as the capabilities of ADAPT change. Three significant changes are in various stages of development as of this writing. The web interface will be updated to use a dynamic platform which will allow a smoother workflow as well as automatic refreshing of data. The web interface update will coincide with an overall enhancement of the status information provided by the web interface. Measures such as the rate of starting and finishing jobs will be plotted for ADAPT overall, for each experiment, and for each computation host. Key diagnostic information will be displayed as well in order to allow the user to mitigate network or hardware limitations on hosts.

The method by which ADAPT currently launches and tracks jobs provides resilience against network and hardware errors. However, at larger scales it may tax the host machine, requiring significant troubleshooting to identify the relevant limitation (e.g., number of files open simultaneously). An updated method of launching and tracking jobs is being developed. This updated method will relay more explicit information about the status of each job to the database while reducing the load on the host machine.

Significant computational resources exist at SNL and other organizations in the form of High Performance Computing (HPC) clusters. A method has been developed by which ADAPT may launch jobs to a set of HPCs through an industry-standard job scheduler (Slurm [41]). This capability has expanded the potential simultaneous running jobs from approximately 130 on a local cluster to 10,000+ on institutional HPCs. Work to integrate this capability into ADAPT will coincide with the revision of the job launching and tracking system.

The plans noted above have resulted from user needs, which are a primary driver of ADAPT development. Requests for additional features or submission of bugs may be directed to ADAPT@sandia.gov.

References

[1] D. M. Kunsman, T. Aldemir, B. Rutt, K. Metzroth, U. Catalyurek, R. Denning, A. Hakobyan, and S. C. Dunagan, "Development and Application of the Dynamic System Doctor to Nuclear Reactor Probabilistic Risk Assessments," Sandia National Laboratories, Albuquerque, NM, SAND2008-4746, May 2008. [Online]. Available: http://prod.sandia.gov/techlib/access-control.cgi/2008/084746.pdf

[2] "Reactor Safety Study: An Assessment of Accident Risks in U.S. Commercial Nuclear Power Plants," United States Nuclear Regulatory Commission, Washington, DC, WASH-1400 (NUREG 75-014), October 1975. [Online]. Available: http://pbadupws.nrc.gov/docs/ML0706/ML070610293.pdf

[3] "Severe Accident Risks: An Assessment for Five U.S. Nuclear Power Plants," United States Nuclear Regulatory Commission, Washington, DC, NUREG-1150, 1990. [Online]. Available: http://www.nrc.gov/reading-rm/doc-collections/nuregs/staff/sr1150/v1/sr1150v1-intro-and-part-1.pdf

[4] R. Chang, J. Schaperow, T. Ghosh, J. Barr, C. Tinkler, and M. Stutzke, "State-of-the-Art Reactor Consequence Analyses (SOARCA) Report," United States Nuclear Regulatory Commission, Washington, DC, NUREG-1935, November 2012.

[5] "Applications of Risk Analysis to Offshore Oil and Gas Operations - Proceedings of an International Workshop," National Bureau of Standards, Gaithersburg, MD, Special Publication 695, May 1985.

[6] M. Stamatelatos and H. Dezfuli, "Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners," National Aeronautics and Space Administration, Washington, DC, NASA/SP-2011-3421 Second Edition, December 2011.

[7] T. Aldemir, "A survey of dynamic methodologies for probabilistic safety assessment of nuclear power plants," *Annals of Nuclear Energy*, vol. 52, pp. 113–124, Feb 2013. [Online]. Available: http://dx.doi.org/10.1016/j.anucene.2012.08.001

[8] N. S. Martin, M. R. Denman, and T. A. Wheeler, "Pruning of Discrete Dynamic Event Trees Using Density Peaks and Dynamic Time Warping," in *Transactions of the American Nuclear Society*, vol. 115. Las Vegas, NV: American Nuclear Society, Nov 2016, pp. 783–786.

[9] L. Humphries, R. Cole, D. Louie, V. Figueroa, and M. Young, "MELCOR Computer Code Manuals - Vol. 1: Primer and User's Guide - Version 2.1.6840 2015," Sandia National Laboratories, Albuquerque, NM, SAND2015-6691R, August 2015. [Online]. Available: https://adamswebsearch2.nrc.gov/webSearch2/view?AccessionNumber=ML15300A479

[10] U. Catalyurek, B. Rutt, K. Metzroth, A. Hakobyan, T. Aldemir, R. Denning, S. Dunagan, and D. Kunsman, "Development of a code-agnostic computational infrastructure for the dynamic generation of accident progression event trees," *Reliability Engineering & System Safety*, vol. 95, no. 3, pp. 278–294, Mar 2010. [Online]. Available: http://dx.doi.org/10.1016/j.ress.2009.10.008

[11] J. LaChance, J. Cardoni, Y. Li, A. Mosleh, D. Aird, D. Helton, and K. Coyne, "Discrete Dynamic Probabilistic Risk Assessment Model Development and Application," Sandia National Laboratories, Albuquerque, NM, SAND2012-9346, October 2012. [Online]. Available: http://pbadupws.nrc.gov/docs/ML1230/ML12305A351.pdf

[12] P. Mattie, R. Gauntt, K. Ross, N. Bixler, D. Osborn, C. Sallaberry, and J. Jones, "State-of-the-Art Reactor Consequence Analyses Project, Uncertainty Analysis of the Unmitigated Long-Term Station Blackout of the Peach Bottom Atomic Power Station," United States Nuclear Regulatory Commission, Washington, DC, NUREG/CR-7155, May 2016. [Online]. Available: http://pbadupws.nrc.gov/docs/ML1318/ML13189A145.pdf

[13] Z. Jankovsky and M. Denman, "Recent Developments in the ADAPT Discrete Dynamic Event Tree Framework," Sandia National Laboratories, Albuquerque, NM, SAND2016-7680PE, August 2016.

[14] L. Humphries, B. Beeny, F. Gelbard, D. Louie, and J. Phillips, "MELCOR Computer Code Manuals - Vol. 1: Primer and User's Guide - Version 2.2.9541 2017," Sandia National Laboratories, Albuquerque, NM, SAND2017-0455O, January 2017. [Online]. Available: https://www.nrc.gov/docs/ML1704/ML17040A429.pdf

[15] R. Gauntt, J. Cash, R. Cole, C. Erickson, L. Humphries, S. Rodriguez, and M. Young, "MELCOR Computer Code Manuals Vol. 1: Primer and Users' Guide Version 1.8.6 September 2005," Sandia National Laboratories, Albuquerque, NM, NUREG/CR-6119 Vol. 1 Rev. 3, 2005.

[16] Z. K. Jankovsky and M. R. Denman, "Modification of the SAS4A Safety Analysis Code for Integration with the ADAPT Discrete Dynamic Event Tree Framework," Sandia National Laboratories, Albuquerque, NM, SAND2017-4764, May 2017. [Online]. Available: http://prod.sandia.gov/techlib/access-control.cgi/2017/174764.pdf

[17] A. Hakobyan, "Severe Accident Analysis using Dynamic Accident Progression Event Trees," Ph.D. dissertation, The Ohio State University, 2006. [Online]. Available: https://etd.ohiolink.edu/!etd.send_file?accession=osu1158672136&disposition=inline

[18] K. Metzroth, "A Comparison of Dynamic and Classical Event Tree Analysis for Nuclear Power Plant Probabilistic Safety/Risk Assessment," Ph.D. dissertation, 2011. [Online]. Available: https://etd.ohiolink.edu/!etd.send_file?accession=osu1306185445&disposition=inline

[19] A. Brunett, "The Assessment of Low Probability Containment Failure Modes Using Dynamic PRA," Ph.D. dissertation, The Ohio State University, 2013. [Online]. Available: https://etd.ohiolink.edu/!etd.send_file?accession=osu1373995939&disposition=inline

[20] D. M. Osborn, "Seamless Level 2/Level 3 Probabilistic Risk Assessment using Dynamic Event Tree Analysis," Ph.D. dissertation, 2013. [Online]. Available: https://etd.ohiolink.edu/!etd.send_file?accession=osu1372524956&disposition=inline

[21] Z. Jankovsky, M. Denman, and T. Aldemir, "A Dynamic Coupled-Code Assessment of Mitigation Actions in an Interfacing System Loss of Coolant Accident," in *Proceedings of the International Conference on Probabilistic Safety Assessment and Management (PSAM 14)*, Los Angeles, CA, September 2018.

[22] B. Rutt, U. Catalyurek, A. Hakobyan, K. Metzroth, T. Aldemir, R. Denning, S. Dunagan, and D. Kunsman, "Distributed dynamic event tree generation for reliability and risk assessment," in *Challenges of Large Applications in Distributed Environments, 2006 IEEE*, June 2006.

[23] A. Hakobyan, R. Denning, T. Aldemir, S. Dunagan, and D. Kunsman, "A Methodology for Generating Dynamic Accident Progression Event Trees for Level-2 PRA," in *PHYSOR-2006, ANS Topical Meeting on Reactor Physics*, Vancouver, Canada, September 2006.

[24] A. Hakobyan, T. Aldemir, R. Denning, S. Dunagan, D. Kunsman, B. Rutt, and U. Catalyurek, "Dynamic Generation of Accident Progression Event Trees," *Nuclear Engineering and Design*, vol. 238, no. 12, pp. 3457–3467, Dec 2008. [Online]. Available: http://dx.doi.org/10.1016/j.nucengdes.2008.08.005

[25] R. Winningham, K. Metzroth, T. Aldemir, and R. Denning, "Passive Heat Removal System Recovery following an Aircraft Crash using Dynamic Event Tree Analysis," in *Transactions of the American Nuclear Society*, vol. 100. Atlanta, GA: American Nuclear Society, June 2009, pp. 461–462.

[26] D. M. Osborn, T. Aldemir, R. Denning, and D. Mandelli, "Seamless Level 2/Level 3 Dynamic Probabilistic Risk Assessment Clustering," in *ANS PSA 2013 International Topical Meeting on Probabilistic Safety Assessment and Analysis*, Columbia, SC, September 2013. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.707.2182&rep=rep1&type=pdf

[27] A. Brunett, R. Denning, and T. Aldemir, "A Reassessment of Low Probability Containment Failure Modes and Phenomena in a Long-Term Station Blackout," *Nuclear Technology*, vol. 186, no. 2, pp. 198–215, 2014.

[28] K. Vierow, K. Hogan, K. Metzroth, and T. Aldemir, "Application of Dynamic Probabilistic Risk Assessment Techniques for Uncertainty Quantification in Generation IV Reactors," *Progress in Nuclear Energy*, vol. 77, pp. 320–328, Nov 2014. [Online]. Available: http://dx.doi.org/10.1016/j.pnucene.2014.04.012

[29] V. Rychkov and K. Kawahara, "ADAPT-MAAP4 Coupling for a Dynamic Event Tree Study," in *ANS PSA 2015 International Topical Meeting on Probabilistic Safety Assessment and Analysis*, Sun Valley, ID, April 2015.

[30] Z. K. Jankovsky, M. R. Denman, and T. Aldemir, "Dynamic Event Tree Analysis with the SAS4A/SASSYS-1 Safety Analysis Code," *Annals of Nuclear Energy*, vol. 115C, pp. 55–72, 2018.

[31] M. Denman, P. Turner, R. Williams, J. Cardoni, and T. Wheeler, "Preliminary Cyber-Informed Dynamic Branch Conditions for Analysis with the Dynamic Simplified Cyber MELCOR Model," in *Transactions of the American Nuclear Society*, vol. 115. Las Vegas, NV: American Nuclear Society, November 2016, pp. 787–790.

[32] Z. Jankovsky, M. Denman, and T. Aldemir, "A Dynamic Assessment of Auxiliary Building Contamination and Failure due to a Cyber-Induced Interfacing System Loss of Coolant Accident," in *International Conference on Topical Issues in Nuclear Installation Safety: Safety Demonstration of Advanced Water Cooled Nuclear Power Plants*, Vienna, Austria, June 2017. [Online]. Available: https://nucleus.iaea.org/sites/gsan/act/CN-251/papers/57-0-Jankovsky_ISLOCA.pdf

[33] A. D. Williams, D. Osborn, K. A. Jones, E. A. Kalinina, B. Cohn, M. J. Parks, E. Parks, B. Jeantete, M. A. Thomas, and A. H. Mohagheghi, "Intermediate Results from a System-Theoretic Framework for Mitigating Complex Risks in International Transport of Spent Nuclear Fuel," in *Proceedings of the 58th Annual Meeting of the Institute of Nuclear Materials Management*, Indian Wells, CA, July 2017.

[34] B. Rutt, A. Hakobyan, K. Metzroth, U. Catalyurek, T. Aldemir, D. Kunsman, and S. Dunagan, "A Software Tool for the Creation and Analysis of Dynamic Event Trees," in *Proceedings of ANS 2006 Winter Meeting*, Albuquerque, NM, Nov 2006.

[35] K. Metzroth, U. Catalyurek, T. Aldemir, D. Kunsman, and S. Dunagan, "A Graphical Tool for the Analysis of Event Trees," in *Proceedings of ANS 2006 Annual Meeting*, Reno, NV, June 2006.

[36] Z. Jankovsky, M. Denman, and T. Aldemir, "Dynamic Importance Measures in the ADAPT Framework," in *Transactions of the American Nuclear Society*, vol. 115. Las Vegas, NV: American Nuclear Society, November 2016, pp. 799–802.

[37] Z. Jankovsky, M. Denman, and T. Aldemir, "Conditional Tree Reduction in the ADAPT Framework," in *Transactions of the American Nuclear Society*, vol. 115. Las Vegas, NV: American Nuclear Society, November 2016, pp. 553–556.

[38] Z. Jankovsky, M. Denman, and T. Aldemir, "Extension of the ADAPT Framework for Multiple Simulators," in *Transactions of the American Nuclear Society*, vol. 115. Las Vegas, NV: American Nuclear Society, November 2016, pp. 557–560.

[39] L. Humphries, D. Louie, V. Figueroa, M. Young, S. Weber, K. Ross, J. Phillips, and R. Jun, "MELCOR Computer Code Manuals - Vol. 3: MELCOR Assessment Problems - Version 2.1.7347 2015," Sandia National Laboratories, Albuquerque, NM, SAND2015-6693R, August 2015. [Online]. Available: https://www.nrc.gov/docs/ML1530/ML15300A476.pdf

[40] L. Kmetyk, "MELCOR Assessment: Gedanken Problems - Volume 1," Sandia National Laboratories, Albuquerque, NM, SAND92-0762, January 1993. [Online]. Available: http://prod.sandia.gov/techlib/access-control.cgi/1992/920762-1.pdf

[41] “SLURM Reference Manual,” Lawrence Livermore National Laboratory, Livermore, CA, UCRL-WEB-201386, 2006. [Online]. Available: https://computing.llnl.gov/LCdocs/slurm/slurm.pdf

A ADAPT Modules

Table A.1: ADAPT Executable Modules

Module	Description
adapt-branch-heritage	Retrieves the parentage of the input branch up to the initial branch of the input experiment
adapt-branch-unexecute	Deletes the child branches and the completed jobs associated with the input branch
adapt-checkhosts	Checks whether the computation hosts in .adaptrc can be reached
adapt-database-admin	Creates, deletes, or backs up the ADAPT database and user as specified in .adaptrc
adapt-editrule-apply	Applies the input BRF and branching condition to the input TSIF. Not typically run by the user
adapt-experiment-checkpoint	Checkpoints the input experiment
adapt-experiment-remove	Deletes the database entries associated with the input experiment
adapt-experiment-restart	Restarts the input experiment from a checkpointed condition
adapt-experiment-update	Updates the probability threshold for the input experiment
adapt-fixprob	Propagates new conditional probabilities for the input branching condition through the input experiment
adapt-gather-outputs	Copies the directories of the input experiment to an input local directory
adapt-job-record	Records the results of the input job number in the database. Not typically run by the user
adapt-load-check	Checks the load on the computation hosts from .adaptrc
adapt-mysql-cleanup	Closes the MySQL connections associated with ADAPT for the user from .adaptrc
adapt-ops	Starts and stops the long-running processes of ADAPT
adapt-pickle-outputs	Gathers the input plot file for all branches in the input experiment into a python pickle file
adapt-plot-horsetails	Produces plot images of the input plot file for the input experiment
adapt-server	Waits for new branches to appear in the queue and runs them when resources are available
adapt-server-halt	Commands adapt-server to stop by writing a stop signal to the database
adapt-status-check	Checks whether adapt-server and adapt-webmin are currently running for the user
adapt-submit	Submits a new branch to the database with the input characteristics. Not typically run by the user
adapt-supervisor	Supervises the running of an ADAPT job on a computation host. Not typically run by the user
adapt-webmin	Provides a web interface to ADAPT

B ADAPT Functions

- ***get_available_executor***(`needed_host`=None)
  Purpose:
  Identifies a host with open capacity, preferring `needed_host` if defined.
  Inputs:
  1. `needed_host`: A desired host

  Returns:
  1. `result`: Returns a host with open capacity

- ***GetDataAccessObject()***
  Purpose:
  Establishes a connection to the ADAPT database
  Inputs:
  1. `needed_host`: A desired host

  Returns:
  1. `get_database_hostname`: Returns the hostname for the database connection
  2. `get_database_port`: Returns the port for the database connection
  3. `get_database_user`: Returns the username for the database connection
  4. `get_database_pass`: Returns the password for the database connection
  5. `get_database_db`: Returns the database for the database connection

- ***CloneDataAccessObject()***
  Purpose:
  Establishes a copy of the ADAPT database connection for a spawned job
  Inputs: None
  Returns:
  1. `Clone()`: Returns the details of the database connection

- ***get_platform()***
  Purpose:
  Queries the operating system being used
  Inputs: None
  Returns:
  1. `result`: Returns the operating system being used

- ***err***(`message`)
  Purpose:
  Writes `message` to `stderr` using a standard format
  Inputs:
  1. `message`: A message to write to `stderr`

  Returns: None

- ***which***(`filename`)
  Purpose:
  Searches for a location containing `filename` in the operating system environment variable `PATH`
  Inputs:
  1. `filename`: A file name to find on the user's `PATH`

  Returns:
  1. `f`: Returns the path if `filename` was found
  2. `None`: Returns nothing if `filename` was not found

- ***requireutil***(`executables`)
  Purpose:
  Writes an error if required `executables` are not found using ***which()***
  Inputs:
  1. `executables`: An executable file that must be located for job success

  Returns: None

- ***unixfind***(`path`)
  Purpose:
  Finds all of the directories and files under `path`, recursively
  Inputs:
  1. `path`: A path to check for directories and files

  Returns:
  1. `out`: Returns the paths of all directories and files found

- ***shell_quote***(`s`)
  Purpose:
  Quotes string `s` in a way that is safe for Unix shells
  Inputs:
  1. `s`: A string to quote

  Returns:
  1. `out`: Return the safely quoted string

- ***shell_unquote***(`s`)
  Purpose:
  Removes quoting in string `s`
  Inputs:
  1. `s`: A string to quote

  Returns:
  1. `None`: Return nothing if the string is only quotes
  2. `out`: Return the unquoted string

- ***shell_escape***(`s`)

Purpose:
Adds escape characters if string `s` contains non-safe characters for Unix shells
Inputs:
1. `s`: A string to escape

Returns:
1. `out`: Returns the escaped string

- ***parsetime***(`timespec`)
Purpose:
Parses user-input string `timespec` for how long to run ADAPT in wall time
Inputs:
1. `timespec`: A time for ADAPT to run. `timespec` may contain one special designator. For example, "30d" represents thirty days

Returns:
1. `int(timespec[0 : len(timespec) - 1])`: If `s` designator is used, assume seconds and return seconds
2. `int(timespec[0 : len(timespec) - 1]) * 60`: If `m` designator is used, assume minutes and return seconds
3. `int(timespec[0 : len(timespec) - 1]) * 3600`: If `h` designator is used, assume hours and return seconds
4. `int(timespec[0 : len(timespec) - 1]) * 3600 * 24`: If `d` designator is used, assume days and return seconds

- ***mkdtempold***(`timespec`)
Purpose:
Makes a temporary directory with a known prefix
Inputs: None
Returns:
1. `file`: Return the name of the directory

- ***load_check***(`host`)
Purpose:
Checks the load average on `host`
Inputs:
1. `host`: A hostname to check for computational load

Returns:
1. `load`: Returns the load average

- ***load_check_all()***
Purpose:
Checks the load average on all ADAPT hosts
Inputs: None
Returns:

1. `unique_hosts, load`: Returns the unique hosts and their load averages

- ***adapt_jobs_check()***
  Purpose:
  Checks the number of jobs running on all ADAPT hosts
  Inputs: None
  Returns:
  1. `host_jobs`: Returns the running jobs on each host

- ***free_adapt_slots***(`method`)
  Purpose:
  Checks all ADAPT hosts for free slots.
  Inputs:
  1. `method`: Defaults to `adapt_jobs_check` but `load_check(host)` may also be used
  Returns:
  1. `free_slots`: Returns the number of free slots on each host

- ***find_files***(`dir`)
  Purpose:
  Finds all of the files under `dir`, recursively
  Inputs:
  1. `dir`: A directory to search for files
  Returns:
  1. `out`: Returns the paths of all files found

- ***get_adapt_d_dir()***
  Purpose:
  Checks the ADAPT configuration file for the directory to place experiment input files
  Inputs: None
  Returns:
  1. `d`: Returns the directory

- ***checkpoint_branch***(`branchid, jobid, checkpointscript, hostname, rundir, pid, wait_completion`)
  Purpose:
  Checkpoints branch `branchid` on a remote computation host by executing the checkpoint script in the job run directory
  Inputs:
  1. `branchid`: The unique branch number
  2. `jobid`: The unique job number
  3. `checkpointscript`: The location of the file used to checkpoint a job
  4. `hostname`: Computation host of the job
  5. `rundir`: Directory of the job

6. `pid`: Process number of the job
7. `wait_completion`: Retries up to 10 times if `True`

Returns: None

- ***kill_remote_pid_group***(`hostname, pid, wait_completion`)
  Purpose:
  Kills process number `pid` on host `hostname`
  Inputs:
  1. `hostname`: Computation host of the process
  2. `pid`: Process number to be killed
  3. `wait_completion`: Retries up to 10 times if `True`

  Returns: None

- ***checkpoint_or_kill_branch***(`branchid, wait_completion`)
  Purpose:
  Checkpoints branch `branchid` if it has not already run. If it is running, kills its job process
  Inputs:
  1. `branchid`: Branch to be checkpointed or killed
  2. `wait_completion`: Retries up to 10 times if `True`

  Returns: None

- ***kill_active_jobs***(`experimentid`)
  Purpose:
  Checkpoints branch `branchid` if it has not already run. If it is running, kills its job process
  Inputs:
  1. `experimentid`: Experiment for which to kill all active jobs. If `None`, performed for all experiments

  Returns: None

- ***experiment_stop***(`experimentid, remove, removefiles`)
  Purpose:
  Stops an experiment, killing all active jobs
  Inputs:
  1. `experimentid`: Experiment to stop
  2. `remove`: Deletes the experiment from the database if `True`
  3. `removefiles`: Deletes files if `True`

  Returns:
  1. `rc`: The return code if any checkpointing scripts were not successful
  2. `None`: If the experiment does not exist
  3. `None`: If removal was not specified

- ***branch_remove***(`branchid_input, remove`)
  Purpose:

Identifies and deletes all database entries associated with the input branch

Inputs:

1. `branchid_input`: Experiment to stop
2. `remove`: Deletes the experiment from the database if `True`, defaults to `False`

Returns:

1. `None`: If the branch does not exist
2. `None`: If removal was not specified

- ***experiment_copy***(`exp, exp_reduced_desc`)

  Purpose:

  Creates a copy of an experiment in the database

  Inputs:

  1. `exp`: An experiment to copy
  2. `exp_reduced_desc`: Description of the copy experiment

  Returns:

  1. `exp`: The experiment that was copied
  2. `exp_reduced_desc`: Description of the copy experiment
  3. `reduced_experiment`: The number of the copy experiment
  4. `new_experiment_row`: The details of the copy experiment from the database

- ***get_branch_heritage***(`branchid`)

  Purpose:

  Finds the parentage of the input branch up to the root branch

  Inputs:

  1. `branchid`: A branch to find heritage for

  Returns:

  1. `out`: The heritage of the input branch

- ***get_exp_branch_heritage***(`experimentid`)

  Purpose:

  Finds the heritage of all end branches of an experiment

  Inputs:

  1. `experimentid`: An experiment to find all branch heritage for
  2. `use_old_data`: Use a cached copy of the heritage if `True`, defaults to `False`

  Returns:

  1. `experimentid`: The input experiment
  2. `root_branch`: The first branch of the experiment
  3. `end_branches`: The end branches of the experiment
  4. `exp_branch_heritage`: The heritage of the all end branches in the experiment

- ***get_plot_data***(`plot_gather_input`)

Purpose:
Reads the input plot file according to the ADAPT plot file format
Inputs:
1. `plot_gather_input`: A tuple containing a host and a file path

Returns:
1. `False`: If the file exists but holds no data in a known format
2. `combined_list`: A list of time series data from the plot file

- ***get_branch_plot_data***(`branchid, plotfile`)
Purpose:
Retrieves plot data for the input file name and branch number
Inputs:
1. `branchid`: A branch number whose job directory should be searched for the plot file
2. `plotfile`: A plot file to read in the branch's job directory

Returns:
1. `False`: If the branch has not finished
2. `False`: If the file exists but holds no data in a known format
3. `combined_list`: A list of time series data from the plot file

- ***get_heritage_plot_data***(`plot_gather_input`)
Purpose:
Retrieves plot data for the input file name and branch number up to the root branch
Inputs:
1. `plot_gather_input`: A tuple containing a branch number and a plot file name

Returns:
1. `plot_values`: Return the time series data

- ***check_jobdir_for_plotfile***(`check_package`)
Purpose:
Checks whether the given file exists in the given branch's job directory
Inputs:
1. `check_package`: A tuple containing a branch number and a plot file name to check

Returns:
1. `branchid`: Returns the input branch
2. `plot_file_meta`: Returns meta information about the plot file or `None` if it does not exist

- ***get_exp_heritage_plot_data***(`experimentid, plotfile`)
Purpose:
Gathers data for the given plot file for all branches in the experiment and assembles them into a dictionary
Inputs:

1. `experimentid`: The experiment to gather plot data for
2. `plotfile`: The plot file to gather
3. `use_old_data`: Use previously-gathered data if `True`, defaults to `False`
4. `use_processors`: Number of processors to use in gathering data, defaults to `1`

Returns:

1. `plot_values`: Returns the time series data for the experiment

- ***json_exp_heritage_plot_data***(`experimentid, plotfile, json_location`)
  Purpose:
  Gathers data for the given plot file for all branches in the experiment, assembles them into a dictionary, and saves the data in a designated location
  Inputs:
  1. `experimentid`: The experiment to gather plot data for
  2. `plotfile`: The plot file to gather
  3. `json_location`: The directory to output a file of the gathered data
  4. `use_old_data`: Use previously-gathered data if `True`, defaults to `False`
  5. `use_processors`: Number of processors to use in gathering data, defaults to `1`

  Returns: None

- ***identify_finished_sequences***(`experimentid`)
  Purpose:
  Identify which end branches in the experiment have finished and which have not
  Inputs:
  1. `experimentid`: The experiment to gather status for
  2. `use_old_data`: Use previously-gathered data if `True`, defaults to `False`

  Returns:
  1. `end_states`: All end branches in the experiment
  2. `finished_sequences`: End branches which have finished
  3. `unfinished_sequences`: End branches which have not finished

- ***get_safe_branches_to_move***(`experimentid, move_commit`)
  Purpose:
  Finds branches in the experiment whose files are safe to move because their immediate children have completed
  Inputs:
  1. `experimentid`: The experiment to check for safe data to move
  2. `move_commit`: Moves data if `True`, defaults to `False`
  3. `remote_server`: A server to copy data to
  4. `remote_user`: A username for copying data
  5. `remote_base_dir`: The base directory on the recipient server
  6. `branch_limit`: The maximum number of branches to move data for at a time

Returns:

1. `safe_to_move`: A list of branches whose data are safe to move

- ***find_var_relevant_branches_and_end_states***(`experimentid, bc_chosen, var_chosen`)
  Purpose:
  Identifies end states in the experiment where the branching condition occurred and tracks values of the ADAPT variable
  Inputs:
  1. `experimentid`: The experiment to check for instances of the branching condition
  2. `bc_chosen`: The branching condition to check
  3. `var_chosen`: The ADAPT variable to track through branching
  4. `use_old_data`: Use previously-gathered data if `True`, defaults to `False`

  Returns:
  1. `relevant_branches`: Branches where the branching condition occurred
  2. `relevant_end_states`: End branches where the branching condition occurred
  3. `relevant_branch_children`: Immediate child branches of branches where the branching condition occurred
  4. `child_branch_values`: Values of the ADAPT variable for immediate child branches

- ***get_branch_variable_history***(`branchid, var_chosen`)
  Purpose:
  Identifies the values of the ADAPT variable for the branch up to the root branch
  Inputs:
  1. `branchid`: The branch to check for values of the ADAPT variable
  2. `var_chosen`: The ADAPT variable to track through branching

  Returns:
  1. `branchid`: The branch to check for values of the ADAPT variable
  2. `var_chosen`: The ADAPT variable to track through branching
  3. `heritage`: The parentage of the branch up to the root branch
  4. `branch_values`: Values of the ADAPT variable for the branch up to the root branch

- ***get_exp_variable_history***(`experimentid, var_chosen`)
  Purpose:
  Identifies the values of the ADAPT variable for all branches in the experiment
  Inputs:
  1. `experimentid`: The experiment to check for values of the ADAPT variable
  2. `var_chosen`: The ADAPT variable to track through branching
  3. `use_old_data`: Use previously-gathered data if `True`, defaults to `False`

  Returns:
  1. `experimentid`: The experiment to check for values of the ADAPT variable
  2. `var_chosen`: The ADAPT variable to track through branching

3. `end_states`: The end branches of the experiment
4. `exp_branch_heritage`: The heritage of all end branches in the experiment
5. `branch_values`: Values of the ADAPT variable for all branches in the experiment

- ***job_remove***(`jobid, commit`)
  Purpose:
  Deletes the job and related entries from the database
  Inputs:
  1. `jobid`: The job to remove
  2. `commit`: Commit to removing the job if `True`, defaults to `False`

  Returns: None

- ***children_identify***(`branchid`)
  Purpose:
  Identifies the immediate children of the branch
  Inputs:
  1. `branchid`: The branch to check for children

  Returns:
  1. `children`: Child branches of the input branch

- ***identify_resulting_end_states***(`branchid`)
  Purpose:
  Identifies the end branches that result from the branch
  Inputs:
  1. `branchid`: The branch to check for eventual end branches

  Returns:
  1. `resulting_end_states`: End branches that stem from the input branch

- ***get_end_branches***(`experimentid`)
  Purpose:
  Identifies all end branches for the experiment
  Inputs:
  1. `experimentid`: The experiment to check for end branches

  Returns:
  1. `end_branches`: End branches in the experiment

- ***get_end_state_progress***(`experimentid`)
  Purpose:
  Identifies the simulation time at which each current end branch in the experiment ended
  Inputs:
  1. `experimentid`: The experiment to check for end branch progress

  Returns:

1. `jobid_times`: End branches and their end simulation times

- ***dynamic_importance_measure_calculation***(`experimentid, bc_chosen, var_chosen, DYI_name, DYI_type, consequence_file, var_value_nonoccurrence, consequence_type`)
  Purpose:
  Calculates dynamic importance measures
  Inputs:
  1. `experimentid`: Experiment to calculate measures for
  2. `bc_chosen`: Branching condition to evaluate
  3. `var_chosen`: ADAPT variable to evaluate
  4. `DYI_name`: Name of the measure to use in displays
  5. `DYI_type`: Type of measure to calculate
  6. `consequence_file`: Plot file to use to measure consequence
  7. `var_value_nonoccurrence`: Value of the ADAPT variable corresponding to non-occurrence of an event
  8. `consequence_type`: Type of consequence measure to use (e.g., `PEAK`, `LAST`, or `ALL`)
  9. `use_old_data`: Use previously-gathered data if `True`, defaults to `False`
  10. `use_processors`: Number of processors to use in gathering data, defaults to `1`

  Returns:
  1. `experimentid`: Experiment to calculate measures for
  2. `bc_chosen`: Branching condition to evaluate
  3. `var_chosen`: ADAPT variable to evaluate
  4. `DYI_name`: Name of the measure to use in displays
  5. `DYI_type`: Type of measure to calculate
  6. `consequence_file`: Plot file to use to measure consequence
  7. `var_value_nonoccurrence`: Value of the ADAPT variable corresponding to non-occurrence of an event
  8. `DYI_value`: Value of the importance measure

- ***pretend_branch_never_completed***(`branchid, commit`)
  Purpose:
  Identify database entries associated with the branch and its children
  Inputs:
  1. `branchid`: The branch to evaluate for undoing completion
  2. `commit`: Deletes the job and all children if `True`, defaults to `False`

  Returns: None

- ***simulate_execution_timings***(`experimentid, nprocs, print_info`)
  Purpose:
  Calculates and prints the progress of the experiment with regards to timing of branch completion and processor use

Inputs:

1. `experimentid`: The experiment to calculate times for
2. `nprocs`: Number of processors to assume for calculations
3. `print_info`: Prints output to `stdout`, defaults to `True`

Returns:

1. `t`: Total time the experiment has run in seconds

- ***simulate_execution_timings_serial_parallel***(`experimentid, nprocs`)
  Purpose:
  Calculates the difference in wall time required between running the experiment with in parallel versus serial
  Inputs:
  1. `experimentid`: The experiment to calculate times for
  2. `nprocs`: Number of processors to use in calculating time differences

  Returns:
  1. `t`: The calculated single-processor wall time as well as actual time required

- ***get_branch_rundir***(`id`)
  Purpose:
  Identifies the job directory for a branch
  Inputs:
  1. `id`: The branch to find the job run directory for

  Returns:
  1. `rundir`: Job directory for the branch

- ***get_branch_hostname***(`id`)
  Purpose:
  Identifies the job host for a branch
  Inputs:
  1. `id`: The branch to find the job host for

  Returns:
  1. `hostname`: Job host for the branch

- ***get_branchid_from_jobid***(`jobid`)
  Purpose:
  Identify the branch associated with a job
  Inputs:
  1. `jobid`: The job to find a branch number for

  Returns:
  1. `branchid`: The branch associated with the job

- ***get_branch_scripts***(`sim`)

Purpose:
Identifies any special scripts associated with a simulator package in the database
Inputs:
1. `sim`: The simulator package name to find scripts for

Returns:
1. `script`: Returns a list of scripts

- ***run_script_for_branch***(`id, script, args`)
Purpose:
Runs a script in the job directory for a branch with given arguments
Inputs:
1. `id`: The branch to run scripts for
2. `script`: The script to be run
3. `args`: Arguments for the script

Returns:
1. `var`: Returns `stdout` from running the script

- ***get_branch_script_file***(`name`)
Purpose:
Identify all special scripts in the database with a given name
Inputs:
1. `name`: The name of the script to search for

Returns:
1. `script`: Returns a list of matched scripts

- ***get_hostname()***
Purpose:
From within a job, get the name of the computation host
Inputs: None
Returns:
1. `hostname`: Returns the host name

- ***get_directory()***
Purpose:
From within a job, get the run directory
Inputs: None
Returns:
1. `directory`: Returns the directory

- ***get_resource_uid()***
Purpose:
From within a job, get the ADAPT resource number of the job execution
Inputs: None

Returns:

1. `resource_uid`: Returns the resource number

- ***job_finished()***
  Purpose:
  From within a job, determines whether the job has completed
  Inputs: None
  Returns:
  1. `status`: Returns `True` if the job is not running and `False` if it is running

- ***job_return_code()***
  Purpose:
  From within a job, identifies the return code of the job command
  Inputs: None
  Returns:
  1. `rc`: Returns the return code of the job command process

- ***job_get_command()***
  Purpose:
  From within a job, identifies the `remotecommand` specified in ***job_start***
  Inputs: None
  Returns:
  1. `jobcmd`: Returns the remote command

- ***job_mark_as_failed()***
  Purpose:
  From within a job, marks the job as failed in the database
  Inputs: None
  Returns: None

- ***has_resources()***
  Purpose:
  From within a job, identifies whether the job host has open capacity
  Inputs: None
  Returns:
  1. `status`: Returns `True` if the host has open capacity and `False` otherwise

- ***make_remote_container***(`name`)
  Purpose:
  Create a job staging directory in the adaptemp directory
  Inputs:
  1. `name`: The name of the directory to create. If `None`, a random name is used
  Returns:
  1. `name`: Returns the directory

- ***cleanup_remote_container***(`container`)
  Purpose:
  Deletes the input job staging directory
  Inputs:
  1. `container`: The name of the directory to delete

  Returns: None

- ***copy_files_to_remote_container***(`files, container, make_executable`)
  Purpose:
  Copies files to the job staging directory and optionally makes them executable
  Inputs:
  1. `files`: The names of files to copy
  2. `container`: The name of the directory to copy to
  3. `make_executable`: Makes files executable if `True`, defaults to `False`

  Returns: None

- ***copy_remote_file_to_local***(`remote, local`)
  Purpose:
  Copies a file from a remote host to the ADAPT server host
  Inputs:
  1. `remote`: The remote file to be copied
  2. `local`: A local path to copy the file to

  Returns: None

- ***copy_remote_file_to_fileobject***(`remote`)
  Purpose:
  Read a remote file to `stdout` on the ADAPT server host
  Inputs:
  1. `remote`: The file to be read

  Returns:
  1. `stdout`: Returns the contents of the file

- ***remote_file_size***(`remote`)
  Purpose:
  Finds the size of a remote file
  Inputs:
  1. `remote`: The file to be evaluated

  Returns:
  1. `sz`: Returns the size of the file in bytes

- ***job_start***(`remotecommand`)
  Purpose:
  Executes a command on the job host

Inputs:

1. `remotecommand`: The command to be run remotely

Returns: None

- ***checkhost***(`ssh, host`)
  Purpose:
  Checks the connection status of a computation host
  Inputs:
  1. `ssh`: The method to use to connect to the host
  2. `host`: The host to be checked

  Returns:
  1. `0`: If the connection was successful
  2. `2`: If key-based authentication is not set up for host
  3. `2`: If incorrect permissions have been set on SSH configuration files
  4. `2`: If the host was not found, possibly indicating a mis-spelled hostname in the ADAPT configuration
  5. `-1`: If the host is known to the network but is down
  6. `-1`: For all other connection failures

- ***get_up_hosts()***
  Purpose:
  Identifies ADAPT hosts that are available
  Inputs: None
  Returns:
  1. `status`: Returns a list of available hosts

- ***get_down_hosts()***
  Purpose:
  Identifies ADAPT hosts that are unavailable
  Inputs: None
  Returns:
  1. `status`: Returns a list of unavailable hosts

- ***get_available_executor***(`needed_host`)
  Purpose:
  Identifies an available host with open capacity, preferring the input host if defined
  Inputs:
  1. `needed_host`: A host to prefer, defaults to `None`

  Returns:
  1. `result`: Returns the name of an available host with open capacity or `None` if there are none

- ***return_executor***(`executor, disable`)

Purpose:

Surrenders the ADAPT resource number of the executor back to the pool and disables it if desired

Inputs:

1. `executor`: The number of the executor to surrender
2. `disable`: Marks the executor as unavailable if `True`, defaults to `False`

Returns: None

- ***get_specific_executor***(`hostname`)

Purpose:

Identifies the connection method used for the current host or `hostname` if defined

Inputs:

1. `hostname`: A host to check, defaults to `None`

Returns:

1. `ssh_executor`: Returns the connection method

C ADAPT Sample Problem Wrapper

A MELCOR ADAPT wrapper is listed below. Note that line numbers are given on the left.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  import os
4  import sys
5  import time
6  import datetime
7  import shutil
8  import subprocess
9
10 getenv_adaptrc = os.getenv('ADAPTRC')
11 getenv_path = os.getenv('PATH')
12 server_path = os.path.join(getenv_adaptrc, 'server')
13 for line in getenv_path.split(':'):
14     sys.path.insert(0, line)
15
16 import adaptlibs
17
18 # BEGIN adapt attributes
19 # checkpoint: melcor-checkpoint
20 # arg1: melcor_root (directory which contains melcor,
      analytic1.rst, etc.)
21 # END adapt attributes
22
23 checkpoint_name = 'melcor-checkpoint'
24
25 jobid = str(os.getenv('NCENGINE_INTERNAL_JOBID'))
26 print('jobid=' + jobid)
27 THISSCRIPT = str(os.getenv('THISSCRIPT'))
28 try:
29     if THISSCRIPT is None:
30         THISSCRIPT = open(os.path.basename(sys.argv[0]), 'r').
              read()
31 except: pass
32
33 this_dir = os.getcwd()
34 stop_word = 'LIZARDKING'
35
36 print('hello, jim jobid %s in %s on %s at %s' % (jobid,
      this_dir, os.uname()[1].split('.')[0], time.asctime()))
37
```

```
38 # Declare files used for this simulator
39 THIS_EXECUTABLE = os.getenv('NCENGINE_EXECUTABLE')
40 MELCOR_ROOT = os.getenv('melcor_root')
41 RST = 'analytic1.rst'
42 SIM1TEMPLATE = 'analytic1.cor.inp'
43 SIM1INPUT = 'analytic1.cor'
44 EDITRULES = 'analytic1_editrules.cor'
45 SIM1EXE = 'melcor'
46 BRANCHESF = 'branches.tmp'
47 OTHERTEMP = 'otherbranches.tmp'
48
49 # Check if this is the root branch.
50 if (str(os.getenv('NCENGINE_ROOT')) == '1') and (str(os.getenv
     ('NCENGINE_RESUMING_CHECKPOINT')) == '0'):
51     print('initializing edit rules for root job at %s' % (time
         .asctime()))
52     shutil.copy2(os.path.join(MELCOR_ROOT, RST), this_dir)
53     shutil.copy2(os.path.join(MELCOR_ROOT, SIM1TEMPLATE),
         this_dir)
54     shutil.copy2(os.path.join(MELCOR_ROOT, EDITRULES),
         this_dir)
55
56     editrule_cmd = 'adapt-editrule-apply --init %s %s 0 melcor
         %s' % (EDITRULES, BRANCHESF, OTHERTEMP)
57     print(editrule_cmd)
58     f = subprocess.Popen(editrule_cmd, shell=True, stdout=
         subprocess.PIPE, stderr=subprocess.STDOUT)
59     f_out = f.stdout.readlines()
60
61     # No loop here, because there is only one branch for the
         initial simulator for the root branch.
62     branchesf_content = open(BRANCHESF, 'r').readlines()[0]
63     shutil.copy2(branchesf_content.split()[0], os.path.join(
         this_dir, SIM1INPUT))
64     os.remove(branchesf_content.split()[0])
65
66
67 if THIS_EXECUTABLE == 'melcor':
68     # Execute MELCOR.
69     print('started executing melcor at %s' % (time.asctime()))
70     print('ls is:')
71     this_dir_list = sorted(os.listdir(this_dir), key=str.lower
         )
72     for listed_file in this_dir_list:
```

```
73          (mode, ino, dev, nlink, uid, gid, size, atime, mtime,
                ctime) = os.stat(str(listed_file))
74          mod_date = datetime.datetime.fromtimestamp(float(mtime
                )).strftime('%B %d %Y %X')
75          print('Name: %s, Size: %s, Modified: %s' % (
                listed_file, str(size), mod_date))
76
77      sim1_exec_command = 'echo E | nice %s %s' % (os.path.join(
            MELCOR_ROOT, SIM1EXE), SIM1INPUT)
78      f = subprocess.Popen(sim1_exec_command, shell=True, stdout
            =subprocess.PIPE, stderr=subprocess.STDOUT)
79
80  # Kill off some branches that are slated to terminate early.
81  if (os.getenv('NCENGINE_TERMINATE_EARLY') is not None) and (
        float(os.getenv('NCENGINE_TERMINATE_EARLY')) > 0.0):
82      time.sleep(float(os.getenv('NCENGINE_TERMINATE_EARLY')))
83      print('terminating early!')
84      open('term-early', 'a').close()
85      open('analytic1.stp', 'a').close()
86
87  # If we want to do anything with stdout/stderr from the
        simulator, such as detect simulator failures, here it is.
88  # This step will wait until the simulator process has closed.
89  f_out = f.stdout.readlines()
90  print('stopped executing %s at %s' % (THIS_EXECUTABLE, time.
        asctime()))
91  print('ls is:')
92  this_dir_list = sorted(os.listdir(this_dir), key=str.lower)
93  for listed_file in this_dir_list:
94      (mode, ino, dev, nlink, uid, gid, size, atime, mtime,
            ctime) = os.stat(str(listed_file))
95      mod_date = datetime.datetime.fromtimestamp(float(mtime)).
            strftime('%B %d %Y %X')
96      print('Name: %s, Size: %s, Modified: %s' % (listed_file,
            str(size), mod_date))
97
98  # If we were checkpointed, stop here.
99  if os.path.isfile('adapt.cp'):
100     print('we were checkpointed at %s on %s, removing
            analytic1.stp' % (time.asctime(), os.uname()[1].split
            ('.')[0]))
101     os.remove('adapt.cp')
102     f = subprocess.call('adapt-checkpoint-note-taken', shell=
            True)
103     sys.exit(0)
```

```
104
105 # Stop here if we don't care about this branch. Early
      termination.
106 if os.path.isfile('term-early'):
107     print('jobid %s terminated early at %s on %s, recording
          result to database' % (jobid, time.asctime(), os.uname
          ()[1].split('.')[0]))
108     f = subprocess.call(('adapt-job-record jobid %s term_early
          1' % (jobid)), shell=True)
109     sys.exit(0)
110
111 if THIS_EXECUTABLE == 'melcor':
112     # Truncate restart file to save space.
113     print('truncating at %s' % time.asctime())
114     SIM1INPUT_2 = 'analytic1.cor2'
115     os.rename('analytic1.mes', 'analytic1.mes.tmp')
116
117     SIM1INPUT_contents = open(SIM1INPUT, 'r').read()
118     SIM1INPUT_contents = SIM1INPUT_contents.replace('''!
          MEL_RFMOD         'newrestart.rst' -1''', '''MEL_RFMOD
                'newrestart.rst' -1''')
119     f = open(SIM1INPUT_2, 'w')
120     f.write(SIM1INPUT_contents)
121     f.close()
122     subprocess.call('echo E | %s %s' % (os.path.join(
          MELCOR_ROOT, SIM1EXE), SIM1INPUT_2), shell=True)
123     os.rename('analytic1.rst', 'analytic1-nontruncated.rst')
124     os.remove('analytic1-nontruncated.rst')
125     os.rename('newrestart.rst', 'analytic1.rst')
126     os.rename('analytic1.mes.tmp', 'analytic1.mes')
127     print('done truncating at %s' % time.asctime())
128
129 # Extracting MELCOR data.
130 extracting_plots = True
131 if (THIS_EXECUTABLE == 'melcor') and (extracting_plots):
132     plot_command = '''\
133         melcor_plot analytic1.ptf analytic1_ separate CVH-P_1
            CVH-P_2 CFVALU_10000 CFVALU_20001 CFVALU_20002
            CFVALU_20011 CFVALU_20012 CFVALU_20021 CFVALU_20022
             CFVALU_99999 \
134         '''
135     f = subprocess.call(plot_command, shell=True)
136
137 if THIS_EXECUTABLE == 'melcor':
138     # Gather some attributes about the completed branch.
```

```
139     melcor_message_file = 'analytic1.mes'
140     melcor_message_file_contents = open(melcor_message_file, '
          r').readlines()
141     for line in reversed(melcor_message_file_contents):
142         if stop_word in line:
143             mystopping_code = line.split(stop_word)[1].split()
                  [0]
144             break
145     for line in reversed(melcor_message_file_contents):
146         if line.startswith(' TIME='):
147             sim_elapsed = float(line.split(' TIME=')[1].split
                  ()[0])
148             break
149     for line in reversed(melcor_message_file_contents):
150         if line.startswith(' Normal termination  TIME='):
151             normal_term = float(line.split(' Normal
                  termination  TIME=')[1].split()[0])
152             sim_elapsed = normal_term
153             break
154
155 # At this point normal_term, sim_elapsed, and mystopping_code
      all MAY exist.
156 # See if we have a branching condition.
157 submitted_bc = False
158 submitted_terminal = False
159 try:
160     f = subprocess.call(('adapt-job-record code %s simtime %s'
          % (mystopping_code, str(sim_elapsed))), shell=True)
161     print('Submitted result of job %s: branching condition %s
          at time %s.' % (jobid, mystopping_code, str(sim_elapsed
          )))
162     submitted_bc = True
163 except:
164     pass
165
166 # See if we have normal termination of melcor for max time.
167 try:
168     f = subprocess.call(('adapt-job-record max_time %s simtime
          %s' % (str(normal_term), str(sim_elapsed))), shell=
          True)
169     print('Submitted result of job %s: maximum problem time %s
          .' % (jobid, str(normal_term)))
170     print('Exiting job %s, as maximum problem time has been
          reached.' % (jobid))
171     submitted_terminal = True
```

```
172 except:
173     pass
174
175 if submitted_terminal is True:
176     sys.exit(0)
177
178 print('submitted_terminal:' + str(submitted_terminal))
179 print('submitted_bc:' + str(submitted_bc))
180 # If nothing else has been submitted, we've failed somehow.
181 if (submitted_terminal is False) and (submitted_bc is False):
182     try:
183         f = subprocess.call(('adapt-job-record logical_fail 1
              simtime %s' % (str(sim_elapsed))), shell=True)
184         print('Submitted result of job %s: simulator (or
              wrapper) failure at time %s.' % (jobid, str(
              sim_elapsed)))
185         print('Exiting job %s, as a failure occurred in either
               the simulator or wrapper. Marking the branch as
              finished.' % (jobid))
186         submitted_terminal = True
187     except:
188         f = subprocess.call(('adapt-job-record logical_fail
              1'), shell=True)
189         print('Submitted result of job %s: simulator (or
              wrapper) failure. No time available.' % (jobid))
190         print('Exiting job %s, as a failure occurred in either
               the simulator or wrapper. Marking the branch as
              finished.' % (jobid))
191         submitted_terminal = True
192
193 if submitted_terminal is True:
194     sys.exit(0)
195
196 # If we reach this line, there is a branching condition to be
     handled.
197
198 # Create child branches and submit them to the database. Check
     $THIS_EXECUTABLE to send correct simulator just run to the
     database.
199 editrule_cmd = 'adapt-editrule-apply %s %s %s %s %s' % (
     EDITRULES, BRANCHESF, str(sim_elapsed), THIS_EXECUTABLE,
     OTHERTEMP)
200 print(editrule_cmd)
201 f = subprocess.Popen(editrule_cmd, shell=True, stdout=
     subprocess.PIPE, stderr=subprocess.STDOUT)
```

```
202 f_out = f.stdout.readlines()
203
204 # Identify the new simulator.
205 branchesf_content = open(BRANCHESF, 'r').readlines()[0]
206 NEWSIM = branchesf_content.split()[7]
207
208 # Submit new branches.
209 branchesf_content = open(BRANCHESF, 'r').readlines()
210 for line in branchesf_content:
211     (config, newstate, branchnum, stopcode, branchhit,
        probability, terminate_early, newsim) = line.split()
212     submit_cmd = ''
213     submit_cmd += 'adapt-submit                    '
214     submit_cmd += '--terminate_early %s        ' % (
        terminate_early)
215     submit_cmd += '--executable      %s        ' % (newsim)
216     submit_cmd += '--handoff         %s %s     ' % (EDITRULES,
        EDITRULES)
217     submit_cmd += '--handoff    %s %s.state    ' % (newstate,
        EDITRULES)
218     submit_cmd += '--handoffref %s %s          ' % (RST, RST)
219     submit_cmd += '--handoffref %s %s          ' % (
        SIM1TEMPLATE, SIM1TEMPLATE)
220     submit_cmd += '--handoff    %s %s          ' % (config,
        SIM1INPUT)
221     if os.path.isfile('analytic1.mes'):
222         submit_cmd += '--handoffref analytic1.mes analytic1.
            mes '
223     submit_cmd += '--probability %s ' % (probability)
224     submit_cmd += 'analytic1                   '
225     submit_cmd += '"sb %s"  ' % (branchnum)
226     submit_cmd += '%s' % (THISSCRIPT)
227     print(submit_cmd)
228     f = subprocess.Popen(submit_cmd, shell=True, stdout=
        subprocess.PIPE, stderr=subprocess.STDOUT)
229     f_out = f.stdout.readlines()
230     os.remove(newstate)
231     os.remove(config)
232     print(f_out)
233
234
235 os.remove(BRANCHESF)
236 print('goodbye, jim jobid %s in %s on %s at %s' % (jobid,
    this_dir, os.uname()[1].split('.')[0], time.asctime()))
```

D ADAPT Sample Problem BRF

A sample set of MELCOR ADAPT branching rules are listed below. Note that line numbers are given on the left. Each line of the BRF is evaluated independently and there is no required order of input.

```
 1 // Name of the simulator template input files with variables
      to be replaced by ADAPT.
 2 InputFile 1 analytic1.cor.inp
 3
 4 // The files used to determine the branching code:
      Stoppingword <simulator> <filename> <magic word> <word on
      line that contains magic word>
 5 StoppingWord 1 analytic1.mes LIZARDKING 2
 6
 7 // The characters used to designate ADAPT variables in the
      simulator template input files.
 8 VarSeparator 1 "{" "}"
 9
10 // The name of the simulators for the database.
11 SimulatorExecutable 1 melcor
12
13 // The simulator to run for the root branch.
14 InitialSimulator 1
15
16 // Initial values not tied to a particular branching condition
      .
17 INIT V99999 FALSE // ADAPT stop CF
18
19 // The valve is initially closed.
20 INIT V10000 0.0
21
22 // The stop for whether to open the valve or not is initially
      armed but not triggered.
23 INIT V20001 0.0
24 INIT V20003 TRUE
25 INIT V20004 FALSE
26
27 BranchingSimulator 20004 1
28 BranchProbability  20004 1 0.5
29 BranchProbability  20004 2 0.5
30
31 // Open or do not open, there is no try.
32 // Do not open.
```

```
33 20004 1 V20003 FALSE
34 20004 1 V20004 FALSE
35 20004 1 V20011 1.e20
36 20004 1 V20021 1.e20
37 20004 1 V99999 FALSE
38 // Do open.
39 20004 2 V20003 FALSE
40 20004 2 V20004 FALSE
41 20004 2 V99999 FALSE
42
43
44 // The stop for valve open time is initially armed but not
      triggered.
45 INIT V20011 10.0
46 INIT V20013 TRUE
47 INIT V20014 FALSE
48
49 BranchingSimulator 20014 1
50 BranchProbability  20014 1 0.25
51 BranchProbability  20014 2 0.25
52 BranchProbability  20014 3 0.25
53 BranchProbability  20014 4 0.25
54
55 // When do we open?
56 // 10s
57 20014 1 V20013 FALSE
58 20014 1 V20014 FALSE
59 20014 1 V20021 10.0
60 20014 1 V99999 FALSE
61
62 // 20s
63 20014 2 V20013 FALSE
64 20014 2 V20014 FALSE
65 20014 2 V20021 20.0
66 20014 2 V99999 FALSE
67
68 // 30s
69 20014 3 V20013 FALSE
70 20014 3 V20014 FALSE
71 20014 3 V20021 30.0
72 20014 3 V99999 FALSE
73
74 // 40s
75 20014 4 V20013 FALSE
76 20014 4 V20014 FALSE
```

```
77  20014 4 V20021 40.0
78  20014 4 V99999 FALSE
79
80
81  // The stop for valve open fraction is initially armed but not
        triggered.
82  INIT V20021 20.0
83  INIT V20023 TRUE
84  INIT V20024 FALSE
85
86  BranchingSimulator 20024 1
87  BranchProbability  20024 1 0.1
88  // BranchProbability  20024 2 0.355
89  BranchProbability  20024 2 0.8
90  // BranchProbability  20024 4 0.355
91  BranchProbability  20024 3 0.1
92
93  // How wide open?
94  // 0.01
95  20024 1 V20023 FALSE
96  20024 1 V20024 FALSE
97  20024 1 V10000 0.01
98  20024 1 V99999 FALSE
99
100 // // 0.25
101 // 20024 2 V20023 FALSE
102 // 20024 2 V20024 FALSE
103 // 20024 2 V10000 0.25
104 // 20024 2 V99999 FALSE
105
106 // 0.50
107 20024 2 V20023 FALSE
108 20024 2 V20024 FALSE
109 20024 2 V10000 0.50
110 20024 2 V99999 FALSE
111
112 // // 0.75
113 // 20024 4 V20023 FALSE
114 // 20024 4 V20024 FALSE
115 // 20024 4 V10000 0.75
116 // 20024 4 V99999 FALSE
117
118 // 0.99
119 20024 3 V20023 FALSE
120 20024 3 V20024 FALSE
```

```
121  20024  3  V10000  0.99
122  20024  3  V99999  FALSE
123
124  TerminateEarly  60  20004  1
```

E ADAPT Sample Problem TSIF

A sample MELCOR ADAPT template input file is listed below. Note that line numbers are given on the left.

```
 1 ! $Id: _M2_GedankenA.inp 6185 2014-08-07 20:02:55Z llhumph $
 2 !Assessment Report
 3 !
 4 !( GlobalData
 5 MEG_DIAGFILE     'analytic1.gdia'
 6 MEL_DIAGFILE     'analytic1.dia'
 7 MEG_OUTPUTFILE   'analytic1.gout'
 8 MEL_OUTPUTFILE   'analytic1.out'
 9 MEG_RESTARTFILE  'analytic1.rst'
10 MEL_RESTARTFILE  'analytic1.rst'  NCYCLE          -1
11 PLOTFILE         'analytic1.ptf'
12 MESSAGEFILE      'analytic1.mes'
13 STOPFILE         'analytic1.stp'
14 EXTDIAGFILE      'analytic1.ext'
15 !MEL_RFMOD        'newrestart.rst' -1
16
17 !)
18 Program MELGEN !(
19 CVH_INPUT !(
20 CV_ID 'CV1-CV1'     1   !CVNAME, ICVNUM, 186 name: CV1 !(
21 CV_TYP 'CVTYPE01' !CVTYP
22 CV_VAT    2      !N      CVZ              CVVOL
23        1 0.0                 0.0
24        2 10.0                100.0
25 CV_THR NonEquil FOG ACTIVE    !ICVTHR, IPFSW, ICVACT
26 CV_PAS COMMON OnlyPool SUBCOOLED    !IPORA, WaterState
27 CV_PTD PVOL 8.E+06
28 CV_PAD 568.23                  !TPOL
29 CV_BND FRAC 1.0                    0.0                    0.0
30 !)
31 CV_ID 'CV2-CV2'     2   !CVNAME, ICVNUM, 186 name: CV2 !(
32 CV_TYP 'CVTYPE03' !CVTYP
33 CV_VAT    2      !N      CVZ              CVVOL
34        1 0.0                 0.0
35        2 100.0               4000.0
36 CV_THR NonEquil FOG ACTIVE    !ICVTHR, IPFSW, ICVACT
37 CV_PAS COMMON OnlyAtm SUPERHEATED    !IPORA, VaporState
38 CV_PTD PVOL 10000.0
39 CV_AAD TATM 568.23
```

```
40 CV_BND FRAC 0.0                    0.0                    1.0
41 !)
42 !)
43 FL_INPUT !(
44 FL_ID 'FL1-FLOWL'     1    !(
45 FL_FT 'CV1-CV1' 'CV2-CV2' 9.9                10.1
46 FL_GEO    0.02          0.2             1.0
47 FL_JSW  3       NoBubbleRise NoBubbleRise ! KFLGFL IBUBF IBUBT
48 FL_SEG    1 !     SAREA              SLEN              SHYD
                   SRGH            LAMFLG    SLAM/CFNAME
49           1    0.2E-01            2.E-01              1.0
                                5.E-05              CONST   0.0
50
51 fl_vlv 1
52         1   'FlowFrac' 'FL1-FLOWL' NoTRIP 'FlowFrac'
53
54 !)
55 !)
56 HS_INPUT !(
57 HS_ID 'HS10001-HS'  10001    !( 186 name: HS
58 HS_GD RECTANGULAR      NO                    ! Type of geometry
      , Steady-state initialization
59 HS_EOD 1.0               1.0                !  HS Elevation and
      Orientation Data
60 HS_MLT     0.50000000E+03                    ! HS Multiplicity
61 HS_SRC NO                                    ! No power source
62 HS_ND    3 ! NXVALU NI XVALUE          TEMPIN          MATNAM
63          1    1     0.00000000E+00    0.56823000E+03 'DUMMY'
64          2    2     0.10000000E-04    0.56823000E+03 'DUMMY'
65          3    3     0.20000000E-04    0.56823000E+03
66 HS_LB CoefTimeTF        'TF2-HTCOEF' 'CV1-CV1'           YES
67 HS_LBP EXT 0.2E-01            9.8E-01
68 HS_LBT 100.0             100.0             1.0
      1.0
69 HS_LBS 1.0 1.0 1.0
70 HS_RB CoefTimeTF        'TF2-HTCOEF' 'CV2-CV2'           YES
71 HS_RBP EXT 0.2E-01            9.8E-01
72 HS_RBT 100.0             100.0             1.0
      1.0
73 HS_RBS 1.0 1.0 1.0
74 HS_FT OFF
75 !)
76 !)
77 TF_INPUT !(
```

```
78  TF_ID  'TF2-HTCOEF'  1.0                    ! 186 name: HTCOEF 186
       numb:       2
79  TF_TAB    4  ! NTFPAR   X                      Y
80                  1     0.0                  1.0
81                  2     50.0                 1.0
82                  3     60.0                 600.0
83                  4     1000.0               600.0
84
85  TF_ID  'TF3-RHO'  1.0                      ! 186 name: RHO 186 numb:
          3
86  TF_TAB    2  ! NTFPAR   X                      Y
87                  1     0.0                  4000.0
88                  2     1000.0               4000.0
89
90  ! *
91  TF_ID  'TF4-CPS'  1.0                      ! 186 name: CPS 186 numb:
          4
92  TF_TAB    2  ! NTFPAR   X                      Y
93                  1     0.0                  10.0
94                  2     1000.0               10.0
95
96  ! *
97  TF_ID  'TF5-THC'  1.0                      ! 186 name: THC 186 numb:
          5
98  TF_TAB    2  ! NTFPAR   X                      Y
99                  1     0.0                  50.0
100                 2     1000.0               50.0
101 !)
102 EXEC_INPUT !( EXEC package start record
103 EXEC_TITLE   'Depressurization (Basecase)' ! Title of the
       calculation
104 !EXEC_GLOBAL_DFT 1.86
105 !)
106 CVH_INPUT    !  CVH package start record
107 ! CVH_SC    1   !SCnumber     Value          Index
108 !           1    4407        1000.0              1
109 MP_INPUT !( MP package start record
110 !*     21 - Number of Materials
       ***********************************************
111 MP_ID   'ZIRCALOY' ! default
112 MP_ID   'ZIRCONIUM-OXIDE' ! default
113 MP_ID   'URANIUM-DIOXIDE' ! default
114 MP_ID   'STAINLESS-STEEL' ! default
115 MP_ID   'STAINLESS-STEEL-OXIDE' ! default
116 MP_ID   'BORON-CARBIDE' ! default
```

```
117 MP_ID   'SILVER-INDIUM-CADMIUM' ! default
118 MP_ID   'URANIUM-METAL' ! default
119 MP_ID   'GRAPHITE' ! default
120 MP_ID   'CONCRETE' ! default
121 MP_ID   'ALUMINUM' ! default
122 MP_ID   'ALUMINUM-OXIDE' ! default
123 MP_ID   'CADMIUM' ! default
124 MP_ID   'STAINLESS-STEEL-304' ! default
125 MP_ID   'LITHIUM-ALUMINUM' ! default
126 MP_ID   'URANIUM-ALUMINUM' ! default
127 MP_ID   'CARBON-STEEL' ! default
128 MP_ID   'B4C-INT' ! default
129 MP_ID   'ZRO2-INT' ! default
130 MP_ID   'UO2-INT' ! default
131 MP_ID   'DUMMY' ! redefined
132 MP_PRTF 4 ! NPAR PROPERTY DEFAULT/TF/CF(may be only for THC)
133              1     ENH       DEFAULT
134              2     CPS       'TF4-CPS'
135              3     THC       'TF5-THC' TF
136              4     RHO       'TF3-RHO'
137 !)
138 !)
139
140
141 CF_INPUT
142
143 ! CF to be changed by ADAPT for open fraction of flow path.
144 cf_id 'FlowFrac' 10000 read
145 cf_sai 1.0 0.0 0.0
146
147 ! Set time to stop for whether valve will open or not.
148 cf_id 'Stp-Alea-Vlv' 20001 read
149 cf_sai 1.0 0.0 1.e20
150
151 ! Trigger for stop for whether valve will open or not.
152 cf_id 'Stp-Alea-Trig' 20002 1-gt
153 cf_liv FALSE
154 cf_arg 2
155        1 exec-time                1.0 0.0
156        2 cf-valu('Stp-Alea-Vlv') 1.0 0.0
157
158 ! Set whether valve opening BC is armed.
159 cf_id 'Stp-Alea-Arm' 20003 1-read
160 cf_liv FALSE
161
```

```
162 ! Stop in the name of love.
163 cf_id 'Stp-Alea-Go' 20004 l-and
164 cf_liv FALSE
165 cf_msg 2 'LIZARDKING 20004 determine whether valve opens or
      not'
166 cf_arg 2
167         1 cf-valu('Stp-Alea-Trig')
168         2 cf-valu('Stp-Alea-Arm')
169
170
171 ! Set time to stop for valve open time.
172 cf_id 'Stp-Time-Vlv' 20011 read
173 cf_sai 1.0 0.0 1.e20
174
175 ! Trigger for stop for valve open time.
176 cf_id 'Stp-Time-Trig' 20012 l-gt
177 cf_liv FALSE
178 cf_arg 2
179         1 exec-time                  1.0 0.0
180         2 cf-valu('Stp-Time-Vlv') 1.0 0.0
181
182 ! Set whether valve timing BC is armed.
183 cf_id 'Stp-Time-Arm' 20013 l-read
184 cf_liv FALSE
185
186 ! Stop in the name of love.
187 cf_id 'Stp-Time-Go' 20014 l-and
188 cf_liv FALSE
189 cf_msg 2 'LIZARDKING 20014 determine valve open time'
190 cf_arg 2
191         1 cf-valu('Stp-Time-Trig')
192         2 cf-valu('Stp-Time-Arm')
193
194
195 ! Set time to stop for valve open fraction.
196 cf_id 'Stp-Frac-Vlv' 20021 read
197 cf_sai 1.0 0.0 1.e20
198
199 ! Trigger for stop for valve open fraction.
200 cf_id 'Stp-Frac-Trig' 20022 l-gt
201 cf_liv FALSE
202 cf_arg 2
203         1 exec-time                  1.0 0.0
204         2 cf-valu('Stp-Frac-Vlv') 1.0 0.0
205
```

```
206 ! Set whether valve fraction BC is armed.
207 cf_id 'Stp-Frac-Arm' 20023 1-read
208 cf_liv FALSE
209
210 ! Stop in the name of love.
211 cf_id 'Stp-Frac-Go' 20024 1-and
212 cf_liv FALSE
213 cf_msg 2 'LIZARDKING 20024 determine valve open fraction'
214 cf_arg 2
215        1 cf-valu('Stp-Frac-Trig')
216        2 cf-valu('Stp-Frac-Arm')
217
218
219 cf_id 'ADAPTCF' 99999 L-OR
220 cf_liv FALSE
221 cf_arg 3
222        1 cf-valu('Stp-Alea-Go')
223        2 cf-valu('Stp-Time-Go')
224        3 cf-valu('Stp-Frac-Go')
225 !
226
227 END Program MELGEN data
228 Program MELCOR !(
229
230
231 CF_INPUT
232
233 ! CF to be changed by ADAPT for open fraction of flow path.
234 cf_id 'FlowFrac' 10000 read
235 cf_sai 1.0 0.0 {V10000}
236
237 ! Set time to stop for whether valve will open or not.
238 cf_id 'Stp-Alea-Vlv' 20001 read
239 cf_sai 1.0 0.0 {V20001}
240
241 ! Set whether valve opening BC is armed.
242 cf_id 'Stp-Alea-Arm' 20003 1-read
243 cf_liv {V20003}
244
245 ! Stop in the name of love.
246 cf_id 'Stp-Alea-Go' 20004 1-and
247 cf_liv {V20004}
248
249
250 ! Set time to stop for valve open time.
```

```
cf_id 'Stp-Time-Vlv' 20011 read
cf_sai 1.0 0.0 {V20011}

! Set whether valve timing BC is armed.
cf_id 'Stp-Time-Arm' 20013 l-read
cf_liv {V20013}

! Stop in the name of love.
cf_id 'Stp-Time-Go' 20014 l-and
cf_liv {V20014}


! Set time to stop for valve open fraction.
cf_id 'Stp-Frac-Vlv' 20021 read
cf_sai 1.0 0.0 {V20021}

! Set whether valve fraction BC is armed.
cf_id 'Stp-Frac-Arm' 20023 l-read
cf_liv {V20023}

! Stop in the name of love.
cf_id 'Stp-Frac-Go' 20024 l-and
cf_liv {V20024}


cf_id 'ADAPTCF' 99999 L-OR
cf_liv {V99999}
!

EXEC_INPUT !(
EXEC_TITLE  'Depressurization (Basecase)'  ! Title of the
   calculation
EXEC_TEND 1000.0              ! End of calculation time
EXEC_STOPCF 'ADAPTCF'
EXEC_TIME    4 !          DTMAX        DTMIN        DTEDT
   DTPLT        DTRST          DCRST
               1 0.0            0.01        0.005          1.0
                             0.01        1000.0        1.E+10
               2 1.0            0.1         0.05           5.0
                             0.1         1000.0        1.E+10
               3 10.0           1.0         1.E-01         500.0
                          2.0         1000.0        1.E+10
               4 1500.0         5.0         1.E-01         1000.0
                        5.0         1000.0        1.E+10
```

```
289 EXEC_CPULEFT  20.0                ! cpu sec left at end of
      calculation
290 EXEC_CPULIM 15000.0             ! Maximum number of CPU seconds
      allowed for this execution
291 EXEC_SOFTDTMIN  1.E-06                  5 ! Define conditions
      for limited under-run of DTMIN
292 EXEC_DTTIME  0.1E-01              ! Initial timestep
293 EXEC_CFEXFILE  'dynamicInp'
294 !)
295 !)
296 END Program MELCOR data
```

F ADAPT Sample Problem Results Analysis

These brief analysis of the ADAPT sample problem (see Section 3.1) is given as an example of the capabilities of ADAPT. DYIs are explored in Section F.1 using the methods introduced in Section 4.3. A tree reduced according to the method from Section 4.4 is presented and briefly analyzed in Section F.2.

F.1 Dynamic Importance Measures

The current DYIs from Section 4.3 were applied to the pressure of `CV001` for sample problem with results shown in Tables F.1 and F.2. The results in Table F.1 are fairly intuitive. Due to the nature of the problem, opening or not opening the valve does not affect the peak pressure seen by `CV001`. The lowest pressure of `CV001` always corresponds with its last pressure. The expected lowest and final values of the pressure when the valve does open are both 0.2075 times the expected pressure when the valve does not open. At the midpoint (approximately 500s), the pressure when the valve opens is 0.2213 times the pressure when the valve does not open.

Table F.1: Example DET DYI1 Values for Valve Opening on `CV001` Pressure

Measure	Value
Peak	1.0
Valley	0.2955
Last	0.2955
Midpoint	0.5683

Table F.2 compares the branched states of valve opening time and open fraction for their effects on the consequence measure. Note that DYI3 compares all values of occurrence. An interpretation of Table F.2 is that when the valve opens at a time of 10s, the final pressure is 0.954 times the final pressure for all sequences where the valve opens. Similarly, a valve open fraction leads to a final pressure 3.28 times the expected pressure across all valve open fractions.

Table F.3 also compares the effects of valve opening time and fraction on the pressure of `CV001`. However, in DYI2 each value of occurrence is compared to non-occurrence. An interpretation of Table F.3 is that when the valve opens at 20s, the midpoint pressure is 0.550 times the pressure in sequences where the valve does not open. For this simple sample problem, the values in Table F.2 and Table F.3 rise or fall monotonically with increasing valve opening time or open fraction. For more complex consequence measures or parameter space, non-monotonic relationships may be seen[18].

[18] See Reference [30] for an application of DYIs to a plant transient DET.

Table F.2: Sample DET DYI3 Values for `CV001` Pressure

Branching Condition	Value	Midpoint	Final
Valve Open Time	10s	0.910	0.954
	20s	0.968	0.983
	30s	1.03	1.02
	40s	1.09	1.04
Valve Open Fraction	0.01	1.74	3.28
	0.5	0.977	0.779
	0.99	0.448	0.489

Table F.3: Sample DET DYI2 Values for `CV001` Pressure

Branching Condition	Value	Midpoint	Final
Valve Open Time	10s	0.517	0.282
	20s	0.550	0.291
	30s	0.587	0.301
	40s	0.619	0.308
Valve Open Fraction	0.01	0.989	0.970
	0.5	0.555	0.230
	0.99	0.255	0.145

DYIs can also be calculated for each time step in each sequence. This calculation is achieved using binning of time steps similar to the method explained in Section 4.2 for calculating the median and mean. Figure F.1 gives the time-dependent DYI1 relationship of the pressure of `CV001` for the event of the valve opening. At the beginning of the analysis, sequences where the valve opens or does not open are relatively close in pressure. As the analysis progresses, they diverge.

Figure F.2 shows the time-dependent DYI2 relationships. Recall that this compares each stated condition (e.g. "valve opens at time 20.0s") to the case of the valve not opening. Over this relatively short time span, sequences where the valve only opens to a fraction of 0.01 give a similar pressure to those where the valve does not open. On the other hand, sequences with a 0.99 open fraction quickly drop to a much lower pressure than those where the valve does not open.

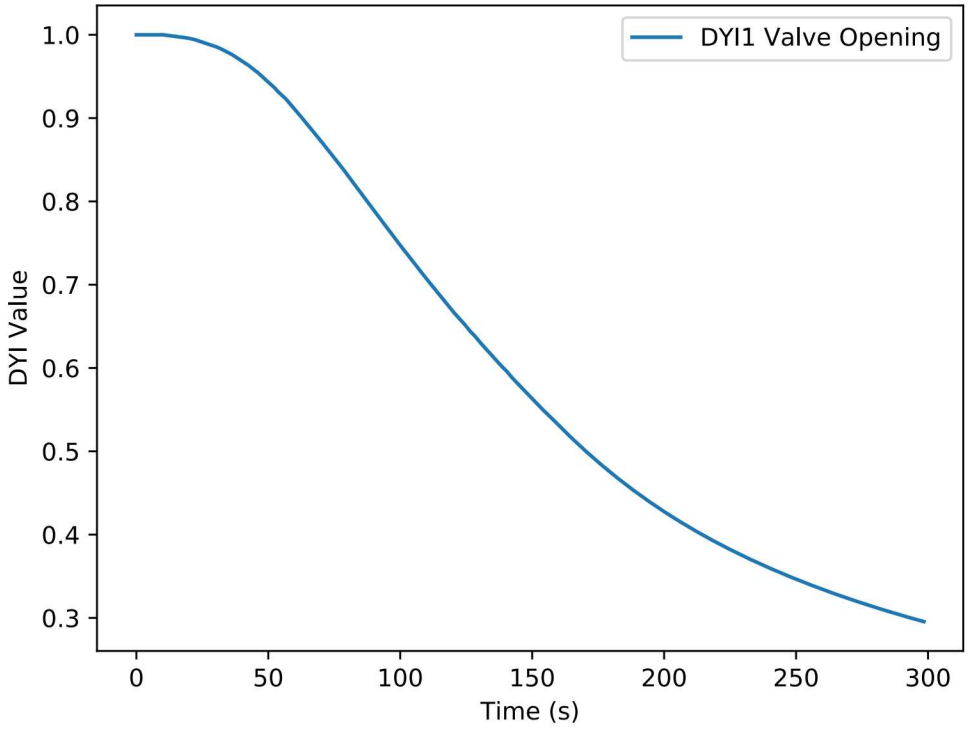

Figure F.1: Time-Dependent DYI1 for Valve Opening and `CV001` Pressure

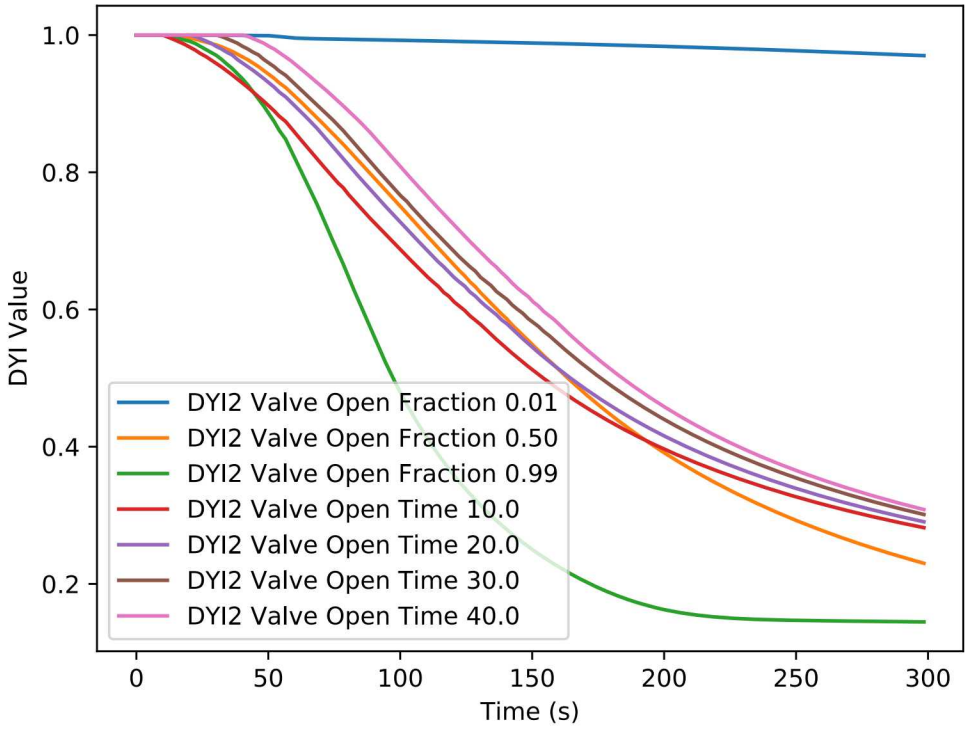

Figure F.2: Time-Dependent DYI2 for Valve Opening and `CV001` Pressure

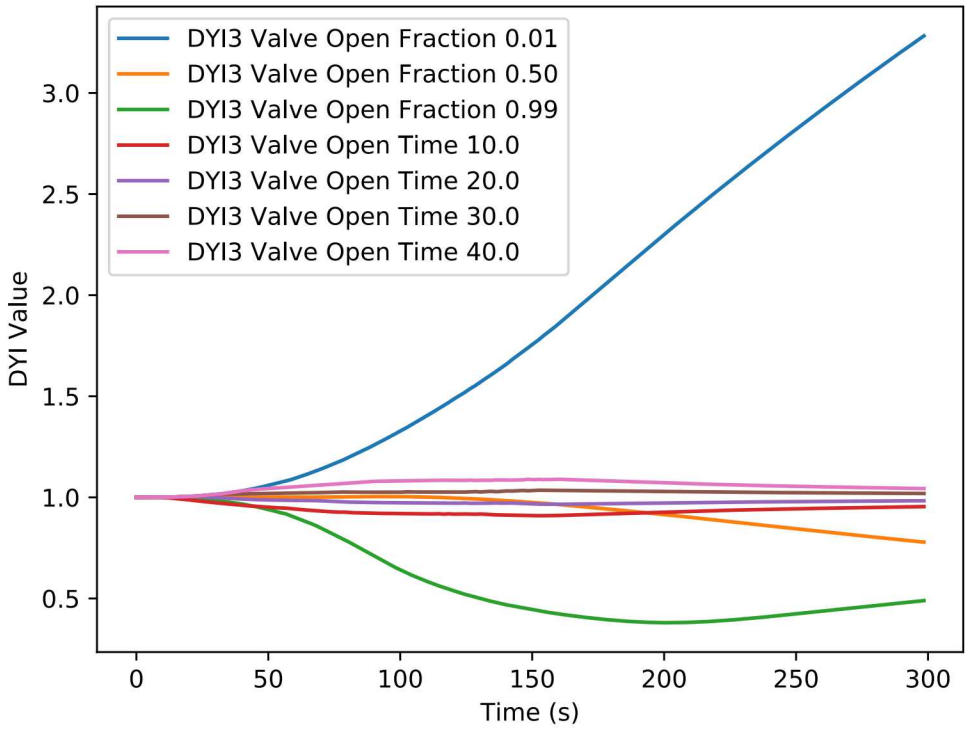

Figure F.3: Time-Dependent DYI3 for Valve Opening and `CV001` Pressure

Figure F.3 shows the time-dependent DYI3 relationships. It can be seen that sequences where the valve opens with a 0.01 fraction diverge from the rest around 50s. This divergence occurs because the pressures of other sequences drop while the pressure in the 0.01 open fraction sequences stays relatively high (see Figure 20). When evaluating time-dependent DYIs, a trend toward a higher value indicates a growing divide between the plotted branch of the input parameter and the others for the chosen consequence measure. In a plant transient analysis this may be helpful in identifying parameters whose values are determined early in the transient but may not have an apparent effect until later.

F.2 Trimmed Tree

The reduced DET from Section 4.4 may be interrogated in the same way as its parent. Quantitative analyses of the reduced experiment may reveal different insights. DYI calculations are performed for the reduced experiment in Table F.4 (compare to Table F.2). Note that in Table F.4 there is no valve open fraction of 0.01 because all sequences with an open fraction of 0.01 were pruned using the rule in Table 9.

Table F.4: Sample DET DYI3 Values for `CV001` Pressure, Reduced by Table 9 Rule

Branching Condition	Value	Midpoint	Final
Valve Open Time	10s	0.891	0.933
	20s	0.962	0.975
	30s	1.04	1.03
	40s	1.11	1.06
Valve Open Fraction	0.5	1.06	1.04
	0.99	0.488	0.656

DISTRIBUTION:

1	MS 0748	Zachary Jankovsky, 8851
1	MS 0748	Troy Haskin, 8852
1	MS 0748	Matthew Denman, 8851
1	MS 0899	Technical Library, 9536 (electronic copy)


Sandia National Laboratories