UNIVERSITY OF CALIFORNIA, SAN DIEGO

Extensions and an explanation module for the iRODS Rule Oriented Verifier

A thesis submitted in partial satisfaction of the requirements for the degree
Master of Science

in

Computer Science

by

Vikram Mavalankar

Committee in charge:

Professor Alin Deutsch, Chair
Professor Yannis Papakonstantinou
Professor Victor Vianu

2008

The thesis of Vikram Mavalankar is approved:

_____

_____

_____
Chair

University of California, San Diego

2008

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

This thesis could not have been written without invaluable help from Dr. Rajasekar Arcot who served as my supervisor and Professor Alin Deutsch who encouraged and challenged me throughout my academic program and patiently guided me through the dissertation process, never accepting less than my best efforts. Many thanks to Richard Liu and Dayou Zhou who have been an incredible support during this whole time. I gratefully acknowledge everyone at the San Diego Supercomputer Center and the UCSD Department of Computer Science and Engineering who have helped me in any small way toward completion of this work.

ABSTRACT OF THE THESIS

Extensions and an explanation module for the iRODS Rule Oriented Verifier

by

Vikram Mavalankar

Master of Science in Computer Science

University of California, San Diego, 2008

Professor Alin Deutsch, Chair

Data grids provide data sharing environments for the management of globally distributed data. Software systems such as iRODS simulate an adaptive middleware architecture that is required for data grids, so that interoperability mechanisms needed to interact with legacy storage systems, as well as the logical name spaces needed to identify files, resources and users, can be controlled. The rule-oriented programming approach is used in the implementation of the iRODS framework wherein management policies are mapped onto rules which can be calibrated by the end user in order to meet existing demands. The Rule-Oriented VErifier (ROVE) framework provides an interface to verify user-specified properties against the underlying iRODS rule base. The actual verification is carried out at the back end by the Web Application VErifier (WAVE) - which is a verification framework for interactive data-driven web applications. The ROVE framework is hence modeled along WAVE web page semantics. The entire process of verification must be performed transparently from end-to-end.

The main focus of this thesis is to analyze the problems associated with the direct application of the existing WAVE explanation module to the ROVE framework. Additionally, this document explains the extensive efforts that have been taken to extend the current capabilities of ROVE so that it can support a more complex underlying iRODS rule-base.

Chapter 1

Introduction

Next generation data handling systems around the world are being designed as data grids to support sharing, publishing, and preserving data residing on storage systems located in multiple administrative domains [10]. A data grid controls sharing and management of large amounts of distributed data. More specifically, it provides logical namespaces for users, digital entities and storage resources in order to create persistent identifiers for controlling access, enabling discovery, and managing wide area latencies. The data grid creates virtual collaborative environments that support distributed but coordinated scientific and engineering research [10, 14].

The Storage Resource Broker (SRB) is a data grid middleware software system designed by the San Diego Supercomputer Center (SDSC) that is operating in many national and international computational science research projects. The SRB creates a data grid based upon the configuration, use patterns, and policies of the underlying system [2]. Further need-based modifications require changes to the SRB code that might introduce unintended side-effects on other operations. These pitfalls led to research and development of a system that

provided greater flexibility by easily adapting to changes in the requirements. Hence, a new adaptive middleware system called iRODS - Integrated Rule Oriented Data System was designed as a glass box where users observe the system functionalities and customize the controls to meet their demands. The power of rule oriented programming has been exploited for achieving the adaptive middleware architecture as required by iRODS. The ultimate goal for iRODS is to delineate management policies and to automate the application of these policies for a vast multitude of data management services. The management policies are mapped onto rules that control the execution of all data management operations.

The Rule Oriented VErifier (ROVE) is a formal verification framework which detects both incorrect or forbidden patterns as well as lack of information prevalent amongst the underlying rule base. ROVE makes use of the Web Application VErifier (WAVE) at the back end. WAVE has been designed to verify a user-defined property in the context of a web-based database application due to which the ROVE framework has been modeled along the template of a web application. The ROVE framework reads in a comprehensive iRODS rule base and invokes the WAVE verifier at the back end to prove or disprove a user-specified property. This verification framework is most useful in determining whether changes to the rule bases would affect desired behavior or violate existing dependency constraints.

The result of the verification is displayed to the user through the 'explanation module' of ROVE. So far, results of the verification demonstrated by the explanation module follow WAVE semantics and are hence out of purview for the ROVE user. Thus, the main problem is the requirement of a translator that translates the verification results presented by WAVE into ROVE semantics. This would help the ROVE user understand the verification results easily. This

thesis focuses primarily on the requirements and design of a comprehensive explanation module for the ROVE user.

The rest of this manuscript is organized as follows: Chapter 2 briefly introduces data grids with examples of the SDSC SRB and iRODS; Chapter 3 describes the architecture and language semantics or iRODS; Chapter 4 discusses the Web Application VErifer and its semantics; Chapter 5 describes the Rule Oriented VErifier; Chapter 6 discusses the extensions made to the ROVE framework and current state-of-art, Chapter 7 distinguishes the requirements and design of an explanation module; and Chapter 8 concludes.

Chapter 2

Data Grids: SRB and iRODS

2.1 Data Grids

The design of today's generic data management systems is threatened by the set of multiple requirements imposed by user communities. In addition, the volume of data is growing exponentially and data sources are distributed across multiple sites, with data being generated in multiple administration domains. The goal of a generic data management system is to build a software infrastructure that can meet the requirements imposed by the user communities. Data grids provide transparency and abstraction mechanisms that applications exploit in order to access and manage data as if they were local to their home system. Moreover, they support virtualization mechanisms for resources, users, and metadata. In summary, data grids provide the required generic data management abstractions needed to manage distributed data [12, 10, 14]. We now concentrate on two specific data grid infrastructures namely, the SRB and iRODS.

2.2 SRB: Storage Resource Broker

An example of a data grid is the Storage Resource Broker (SRB) developed at the San Diego Supercomputer Center. The SRB manages context (administrative, descriptive, and preservation metadata) about content or digital entities such as files, URLs, SQL command strings, directories. The content may be distributed across multiple types of storage systems across independent administration domains.

By separating the context management from the content management, the SRB easily provides a means for managing, querying, accessing, and preserving data in a distributed data grid framework [2, 9]. SRB abstracts data object names, resources, users and groups, and provides uniform methods for dealing with them. SRB hides the underlying physical infrastructure from users by providing global, logical mappings to the digital entities registered into a shared collection. Hence, the peculiarities of storage systems and their access methods, the locations of data, user authentication and access across systems, are hidden from the users. Hence, a user can access files from an online file system or the web without worrying about their location and other connection details. Thus, SRB provides a grid-level middleware for sharing data and metadata distributed across heterogeneous resources using uniform APIs and GUIs.

2.3 iRODS: An Integrated Rule-Oriented Data System

iRODS is a data grid software system built on the foundation of its predecessor, SRB. iRODS extends the architecture of SRB in order to achieve a higher

level of virtualization. The virtualization of policy and constraints was not seen before in SRB and has been implemented in the iRODS framework. SRB implements internal consistency constraints within itself as part of the hard-coded infrastructure. This lacks a much-required flexibility as SRB internal code structure is difficult to modify, thus users are often forced to design policies that cater to SRB functionalities.

iRODS goes a step further and seeks to strengthen the areas of flexibility and customizability. This is implemented by having the functionalities made to fit user-designed policies instead. In other words, users observe the system functionalities and customize the controls to meet their demands [13, 8]. This migration from a primitive black box philosophy - like that used in SRB, to a transparent glass box philosophy - like that used in iRODS is a major design objective for iRODS.

Regular middleware systems are designed around the black box philosophy in which the flow of operations are immutable programmatically, except configuration changes that may allow one to set the initial environment conditions of the middleware. The transparent glass box philosophy, in contrast, forms the basis for the design of an adaptive middleware system. There are multiple ways for achieving adaptive middleware architecture. In the iRODS approach, the power of Rule Oriented Programming (ROP) has been exploited. In ROP, the power of controlling the functionality rests more with the users than with system and application developers. Hence, any change to a particular process or policy can be easily constructed by the user and tested and deployed without the aid of system and application developers.

The final design goal for iRODS is to distinguish management policies and to automate the application of these policies for a multitude of data management services. These management policies are mapped onto rules that con-

trol the execution of all data management operations. This provides a means for encoding customization of data management functionalities in an easy and declarative fashion using Rule Oriented Programming.

The next chapter focuses on the architecture of iRODS for further clarity. The language semantics used by the iRODS framework for specifying rules, actions and other system functionalities are described in detail.

Chapter 3

iRODS: Architecture and Semantics

3.1 Architecture

The main design concept as mentioned earlier in the iRODS framework is the rule oriented approach. Data management and control functionalities are provided in a declarative and user-friendly manner via the rule oriented programming model [13, 8]. System functionalities being performed in the iRODS data grid system are coded as 'rules'. When a rule is invoked, it explicitly signifies the execution of a system operation that performs a particular task. These operations are called micro services in iRODS. Micro services, in other words, are simple programming language level function calls that are executed while reading the rule body. While executing the rules, the flow of tasks can be modified in different ways. For example, new micro-services could be interposed in a given rule or the micro service code could be modified appropriately and recompiled. Further, a new rule with a higher priority could be added in the rule base in place of another rule for the same task, so that this new rule gets chosen before the existing rule. The importance of this additional level of redundancy

is significant. The new pre-emptive rule will be executed before the original rule and so, if there is a failure in the execution of any part of this new rule then the original rule gets executed.

There are three main design concepts of the iRods architecture. Firstly, the iRODS data grid architecture is based on a client/server model. Secondly, the iRODS distributed storage is a database system for maintaining operations, attributes and states of data. The third and significantly important design concept is the support of a comprehensive rule based system for enforcing and executing adaptive rules. The main focus of this chapter is to introduce the language semantics primarily used by the iRODS rule system. Before this, it is important to understand the iRODS rule system and associated components.

3.2 Rule System

The iRODS rule system comprises of an iRODS rule engine at the kernel of the framework. This rule engine runs on all iRODS servers and is responsible for invoking a number of predefined micro services. The rule being executed is interpreted and the corresponding micro service invocation is invoked by the rule engine.

There are different kinds of rules based on the high level operation that they are meant to perform. For example, system level rules are responsible for data management policies and automation of system level services. System level rules can be invoked on the servers internally to enforce and execute management policies for the system. Another class of rules allows users to request the iRODS servers to perform a sequence of operations on behalf of the user. This is done when clients externally invoke the rule engine by a prede-

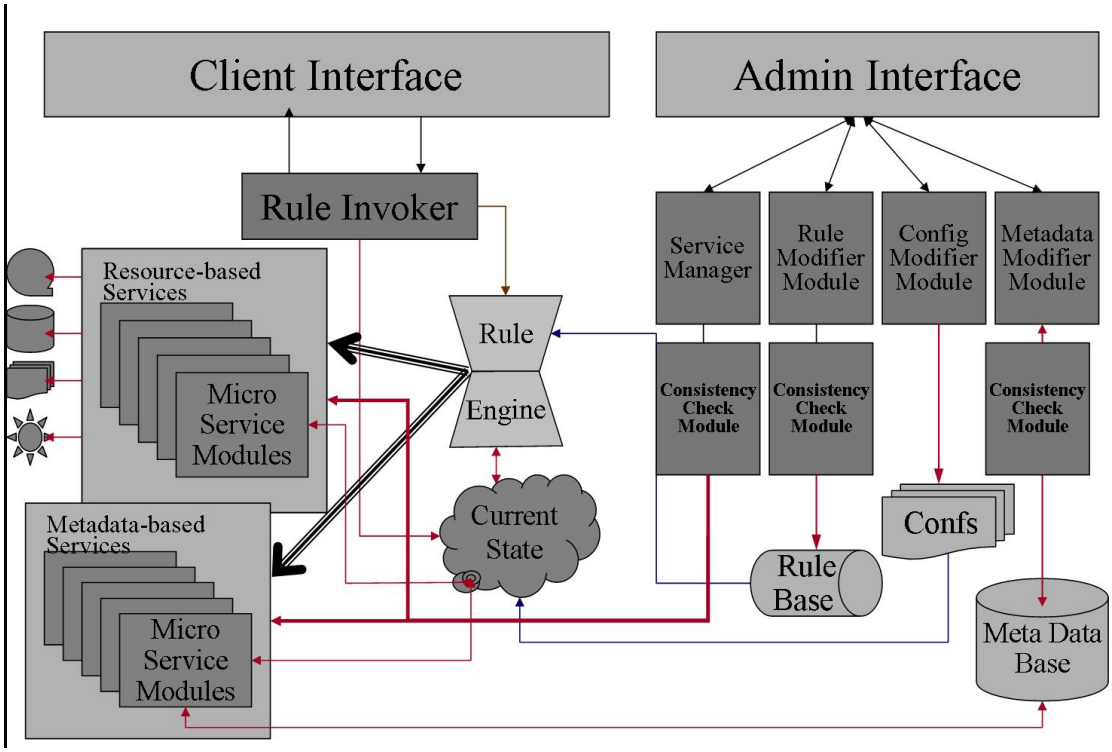

Figure 3.1: iRODS Architecture

fined command (such as irule) or API. Some rules require immediate execution while others may be executed at a later time in the background. The iRODS rule system supports both these kinds of rules.

The iRODS architecture and its various components can be shown graphically in Figure 3.1. The two main components are the client interface and the administrator interface. The client interface middleware follows the conventional 'black box' design philosophy in which the user utilizes the methods of the rule invoker. These methods of the rule invoker are instrumental in invoking metadata-based and resource-based services. The services in turn are invoked by the rules and are responsible for updating the metadata or the resources. Rules are stored collectively in iRODS rule base (IRB) files.

The second main component of the iRODS architecture is the administrator interface. The administrator interface follows the 'glass box' design philosophy. This is implemented by having various modifier modules that permit the administrator to customize the functionality of the software without directly modifying the code. For example, the user may alter policies via the rule modifier module. Since there is a scope for the user to change policies by altering the rules, there needs to be a consistency-checking module to eliminate undesirable side effects as a result of modifying the rules. In other words, alterations to the rules must not violate desired restrictions and the consistency module is thus used to verify the syntax and semantics of the rule-oriented specification language.

3.3 Virtualization in iRODS

The iRODS framework allows the virtualization of policy and constraint management. Alternatively, it can be viewed as a way of providing a new abstraction for the data management processes and policies themselves. Data management policies are mapped onto rules that control the execution of all data management operations. iRODS supports management policy virtualization. The rules can be implemented independently of the remote storage system. For each desired outcome, corresponding rules are defined. These rules control the execution of the standard remote operations. The operations that are performed by the rule-based data management systems can be abstracted in terms of micro-services. A logical name space can be constructed for the micro-services. The next section describes the semantics of iRODS micro-services. The logical namespaces make it possible to organize micro-services without having to change the management policies. Thus iRODS supports service virtualization. Similar strategies help support rule virtualization.

3.4 iRODS Micro-services

When a rule is invoked, it explicitly signifies the execution of a system operation that performs a particular task. This operation is called a micro-service in iRODS. Micro-services, in other words, are well-defined procedures that perform a certain task. They are simple programming language level function calls that are executed while reading the rule body. Micro-services are developed and made available by programmers and compiled into the iRODS server code. Users and administrators can interleave these micro-services to implement a higher macro-level operation. While executing the rules, the flow of tasks can be modified in different ways. For example, new micro-services could be interposed in a given rule or the micro service code could be modified appropriately and recompiled. Further, a new rule with a higher priority could be added in the rule base in place of another rule for the same task, so that this new rule gets chosen before the existing rule. Using priorities and validation conditions, at run-time, the system chooses the 'best' micro-service chain to be executed.

Micro-services can perform an operation that can be quite small or very involved. It is up to the micro-service developer to choose the appropriate level of granularity for distinguishing between operations. Conventionally, a large operation can be divided into sub-tasks with well-defined interfaces and each of these sub-tasks can be defined as a micro-service. If two such sub-tasks are always executed together, it would be best to group them into one micro-service. On the other hand, defining a large operation as a single micro-service takes away the control that is given to the end user/administrator who might want to choose not to do some portions of the operation.

3.5 iRODS Actions

As mentioned in the previous section, system users can interleave micro-services to implement complex macro-level functionalities. These macro-level functionalities are called actions. It is possible to have more than one chain of micro-services for an action. In other words, a system can have multiple ways of performing the same action and using priorities and validation conditions, at run-time, the system chooses the 'best' micro-service chain to be executed.

In the iRODS architecture, collections of rules that pertain to data grid management and manipulation of modules reside in the iRODS Rule Base (IRB) files. On a higher level, the macro-level functionalities of the system are represented by actions. Every rule in an IRB is a particular definition of an action. In other words, an action is the name given to a rule. This is synonymous to the head atom in a Prolog rule, or trigger-name in a relational database. Each action can consist of a chain of other actions or micro-services. The next section describes the rule semantics used for defining actions and micro-services.

3.6 iRODS Rule/Action Semantics

Rules are used for action definitions and consist of micro-services and other actions. The iRODS server has a built-in rule engine that interprets rules and invokes the appropriate micro-services as needed. The rules are stored in iRODS Rule Base (IRB) files. There can be more than one IRB file. The administrator can include more than one rule base file and in this case, the files will be read in order.

A rule is specified as a regular line of text that contains four parts sepa-

rated by the '|' separator:

```
actionName | condition | workflow-chain | recovery-chain
```

- '*actionName*' is the name of the rule. It is an identifier which can be used by other rules or external functions to invoke the rule.
- '*condition*' is the condition under which this rule applies. i.e., this rule will apply only if the condition is satisfied.
- '*workflow-chain*' is a sequence of micro-services or rules that are executed by this rule. They are separated by the '##' separator. Each may contain a number of input/output parameters.
- '*recovery-chain*' is a sequence of micro-services or rules that must be invoked when execution of any one of the micro-services or rules in the workflow-chain fails. There should be an equal number of micro-services or rules in the recovery-chain as there are in the workflow-chain. Note that if any sub-task in the workflow-chain fails, then the entire set of changes made by the rule invocation must be undone. If no recovery action is needed for a given micro-service or rule, a 'nop' action should be specified.

The next section analyzes and describes what happens when iRODS executes a rule.

3.7 iRODS Execution Environment

The execution of a rule in iRODS translates to the execution of a chain of micro-services. Although there might be actions in the chain, from the pre-

vious sections it is clear that these actions can be flattened to a list of underlying micro-services. Thus, the fundamental atomic unit in rule execution is the micro-service execution. To better understand micro-service execution, it is useful to understand the iRODS execution environment.

The data components of iRODS can be divided into three categories:

- *Session Attributes* - Each running instance of iRODS has its own set of session attributes. These attributes have an external namespace and are mapped onto internal data structures. The session (denoted as '$') represents the transient memory of iRODS.
- *Database Attributes* - Like the session attributes, database attributes also have an external namespace which is mapped onto internal database schema. The database (denoted as '#') represents persistent storage that is shared across all sessions.
- *Side Effect Set* - The side effects (denoted as '%') represent changes to the external environment, such as the creation of a file or the sending of a message.

When a micro-service is executed, the following occurs:

- A value of a session variable ($) gets modified in the temporary session memory. This affects other rules and micro-services executed in the same session.
- A row in the database gets inserted, deleted or modified. This also affects the current and subsequent sessions.
- An operation outside the iRODS system is performed. For example, file creation, modification or deletion in a remote file system.

3.8 iRODS Rule Engine Execution Workflow

This section examines the actions performed by the iRODS rule engine. When the rule engine is presented with an action name, it selects all the rules whose action names are the same. These rules are prioritized based on the order in which they were read into the rule base of the rule engine. The next phase is condition validation. The first rule in the ordered list is checked for validation of its condition. If the condition fails, then the next rule is tried. If no more rules are available then the action fails and a failure status (negative number) is returned to the calling routine.

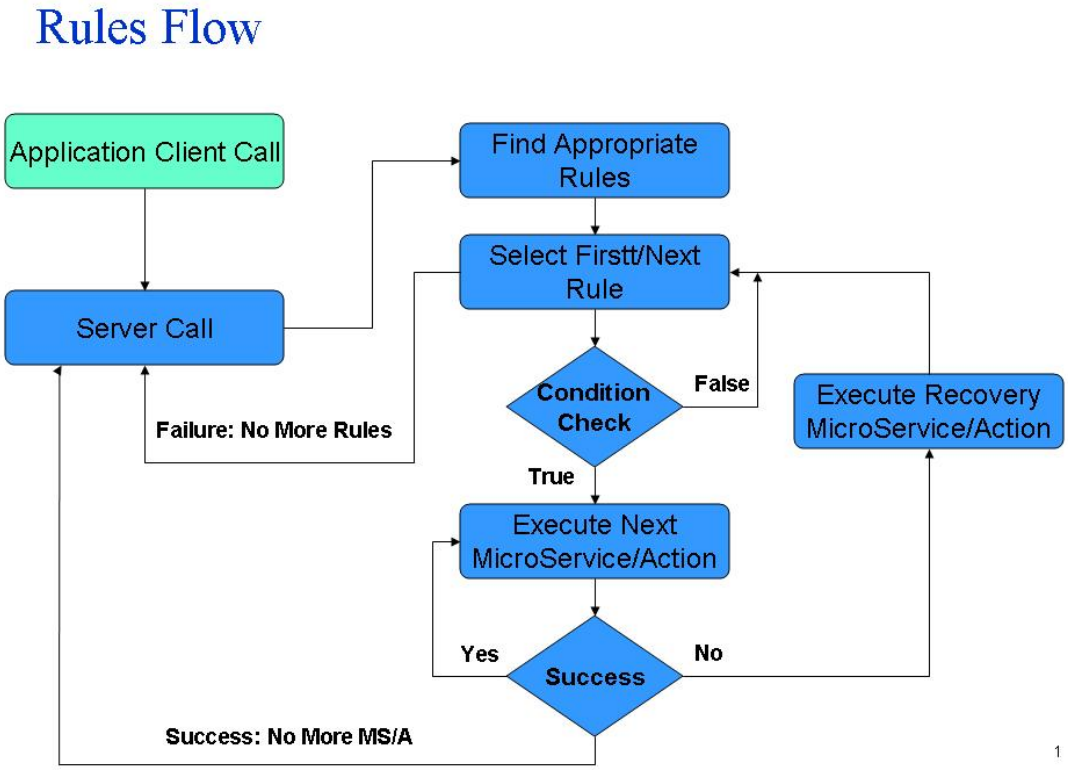

Figure 3.2: iRODS Rule Engine workflow

On the other hand, if the condition succeeds, then the micro-services and action chain in the rule are executed orderly from left to right. If all of the micro-services succeed then the action is considered a success and a success status (0) is sent to the calling routine. In the event of a failure of one of the micro-services/actions in the chain, the rule engine starts a recovery procedure. It applies the corresponding recovery procedure that is defined in the rule. The recovery for the failed micro-service/action is first performed, followed by the recovery of all the previously successful micro-services/actions in reverse order. At the end of recovery, the values of the session variables ($), database variables (#) and side-effect set (%) are rolled back.

This entire process is shown in Figure 3.2.

3.9 Extended Normal Form for Micro-services

The execution of micro-services can be encapsulated as a the execution of a flow of input-output relationships. The data components which are modified can be viewed as the output and the previous changes can be viewed as the input. It is clear from the previous sections that at the lowest level, all actions are composed of micro-service calls. Hence, studying the input-output relationships for the micro-services would be significantly helpful in understanding the semantics of the macro-level actions. In order to achieve this, it is necessary to describe the micro-services in an abstract form. The normal form language for iRODS rule specification currently does not support the abstraction of a micro-service. Hence, for the purpose of verification, an extension to the existing language has been proposed so that the micro-services can be decomposed into even smaller components such as their individual effects.

The new abstraction enables micro-services to be written in the form of rules, in order to abstract the behavior of the micro-service. It demonstrates the effect of the micro-service on data components such as the transient session, the persistent database, and other side effects. The existing normal form language has been extended instead of creating a new language for a variety of reasons. For example, it is easier for the verification framework to create pseudo rule bases that contain rules of both formats, which could be parsed and translated with a single parser.

A rule for a micro-service, just like that for an action, is specified as a regular line of text that contains three parts separated by the '|' separator:

```
microserviceName | Condition | Effects
```

- *microserviceName* - The name or unique identifier of the micro-service. Unlike actions, there can be only one definition for each micro-service.
- *Condition* - The condition that must hold for the micro-service to execute. This condition, like the condition for an action, applies to the entire body.
- *Effects* - The set of effects produced as a result of successful execution of the micro-service. (Separated by '##')

Unlike actions, micro-service rule definitions do not have a recovery segment. This is because it is assumed that an alternative micro-service exists to unroll the effects.

The following chapter introduces the Web Application VErifier (WAVE) that is used for the verification phase of the iRODS framework.

Chapter 4

WAVE: Web Application VErifier

The Web Application VErifier (WAVE) is a framework for high level specification and verification of interactive, data-driven web applications [5, 3]. Verification of data-driven web applications, in particular, is important as it leads to further assurance in the preciseness of the web applications. WAVE was designed and developed by the Database Group at the Department of Computer Science and Engineering of University of California, San Diego. WAVE not only provides a comprehensive verification framework, but more specifically, WAVE can prove whether a property formula is observed throughout all runs of the web application, given the various possibilities of user input and page transitions. A property formula in this case would pertain to either or both the status of the web application and the state of the underlying database.

The correctness of the iRODS framework can be enhanced significantly by verification of properties such as checking appropriate user permissions required to invoke certain micro-services. WAVE is complete for a broad class of applications and temporal properties. For other applications, WAVE can be used as an incomplete verifier, as commonly done in software verification

[5, 4, 6]. Experimental results on WAVE demonstrated that the verification times were in the order of seconds. Hence, interactive applications controlled by database queries (such as iRODS) may be well suited to automatic verification. The experimental results obtained also illustrate the effectiveness of the unification of model checking with database optimization techniques as used in the implementation of WAVE.

4.1 WAVE Architecture

The main workflow of the WAVE framework is that it takes the specifications of a web application and a property as the input, and outputs whether the specified property is true or false. In case the property evaluates to false, WAVE displays a counter-example which is a run of the web application violating the property [7]. There are four main components or modules that form the core of the WAVE framework. These modules interact with each other as shown in Figure 4.1 and their individual functionalities are described below:

- *Specification module*

  This module is responsible for representing or modeling the web application in the form of a web page schema. On a higher level, the web page schema specifies the input options available to the user at each page of the web application. Further, it also specifies how the state of the underlying database or environment would be modified as the user transitions through the various pages of the web application. More specifically, a page schema specifies user inputs accepted at the current page, as well as the logic using a collection of rules governing input options generated for the user. Once the user supplies an input, the rules specify subsequent state modifications; actions performed and transition information to the

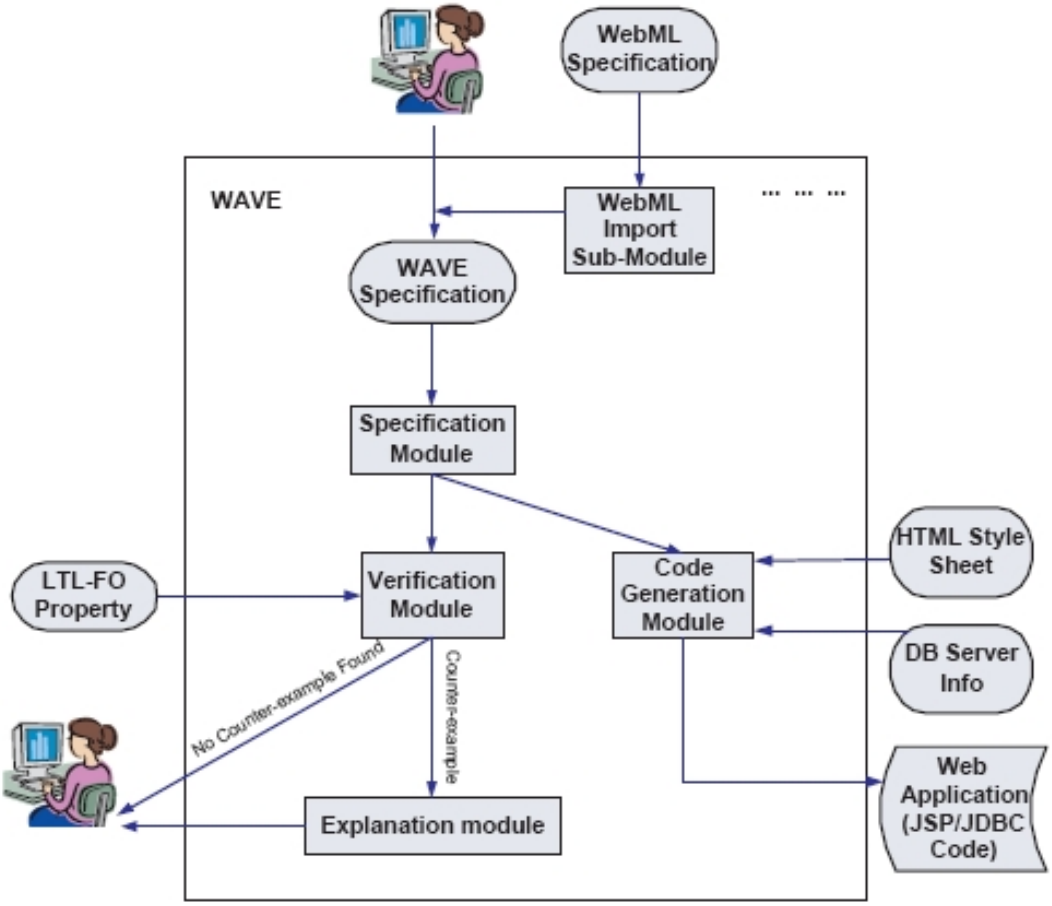

Figure 4.1: WAVE Architecture

next page. The web schema specifications are stored in a specification file which conforms to a grammar. The WAVE parser reads this specifications file and generates an internal representation of the specification, which is consumed by the verification module and the code generation module.

- *Verification Module*

  The verification module performs the function of searching for a counter-example run or a run that violates the specified property. To generate the counter example requires examining all the underlying databases which

is not feasible. It has been demonstrated in [5] that it is sufficient to consider a finite set of constants for values in the database, yielding a fixed number of databases. WAVE executes runs based on each database and for each run, the user specified property is checked and violations are detected. The final result is a true or false truth value for the property along with a counter-example run in the case of a false truth value for the property.

- *Explanation module*
  The explanation module is responsible for conveying the results of the verification to the user in an understandable manner. If the specified property evaluates to false, the counter example generated shows the exact sequence of user inputs and corresponding evolution and point of failure of the web application. This thesis focuses primarily on the needs and requirements for an explanation module when utilizing the WAVE verifier for the iRODS framework.

- *Code generation module*
  If the verification completes successfully, then the specification is used to automatically generate the code implementing the web application. It is the responsibility of the code generation module to read in the specification file and generate JSP pages which can be deployed on a Tomcat server in order to be viewed in a browser.

4.2 Web Application Specification Schema

This section describes the semantics used by WAVE for specification of interactive, data-driven web applications. On a higher level, external users or programs supply the required input to the web application. On receiving this

input, the application responds by carrying out an action such as updating its internal state database and moving to a new web page. The web application generates web pages dynamically by queries on an underlying database.

More specifically, the web application is modeled in the form of a web page schema describing the structure of the web pages as viewed by the user at any point of time. The web page schema specifies the input options available to the user at each page of the web application and how the state of the underlying database would change as the user transitions through various pages of the web application. All inputs, actions, states and the database are modeled as relations.

A typical web page schema for WAVE has the following components:

- **Static schema**
  - *Page* - A page represents each web page in the schema. Pages are identified by a unique name. A 'homepage' is specified as the starting point of the application.
  - *Input options* - They specify which input options are available to the user at the current page and the conditions that each input option can be selected under.
- **Transitions**
  - *Target Webpage* - Each page must have a set of target pages approachable from it by certain user actions.
  - *State* - All modifiable values of the web application are represented as states. A state is essentially a relation table.
  - *Actions* - The actions represent all the actions performed by the page under consideration.

For example, the specification of a home page for an online bookstore site might look like the specification as outlined below. The online bookstore site would sell books and music CDs to its customers.

```
 Page:  Home-page(HP)
Input:
     clickbutton(x);
Input Rules:
     Option@clickbutton(x) := x= "home" or x="book"
     or x="music" or x="view cart";
State Rules:
     not home():= clickbutton("book") or
     clickbutton("music");
     book():= click("book");
     music() := clicknutton("music");
Action Rules:
Target Webpages:  BBP, BMP, CAP, MWP
Target Rules:
     BBP := clickbutton("book");
     HP := clickbutton("home");
     BMP := clickbutton("music");
     CAP := clickbutton("view cart");
```

In the schema above,

- The pages are:
  BBP - 'Buy a book' page
  HP - 'Home page'

BMP - 'Buy a music CD' page
CAP - 'View cart' page

- The input option is:
  Click Button - This represents the various buttons that can be pressed by the user. As the specification shows, there are four possible options (Home, Book, Music, View cart) that when clicked on, would invoke the corresponding target pages.

- The action and transition rules are:
  The target webpage is dependent on which button was pressed by the user.

4.3 Property Specification

Along with the specification of a web application, WAVE also takes a user specified property as the input, and outputs whether the specified property is true or false. In case the property evaluates to false, WAVE displays a counter-example which is a run of the web application violating the property. A run is a sequence of configurations through which the web application evolves in response to user inputs. Examples of properties can range from basic soundness of the specification to semantic properties. Such properties are expressed using LTL-FO, an extension of linear-time temporal logic (LTL) [1, 6]. For example, the LTL-FO formula

$$\forall x \forall y \forall id \left[\left(pay\left(id, x, y\right) \wedge price\left(x, y\right) \right) \ \mathbf{B} \ ship\left(id, x\right) \right]$$

states that whenever item x is shipped to customer id, a payment for x in the correct amount must have been previously received from customer id.

Chapter 5

ROVE: Rule Oriented VErifier

The Rule Oriented VErifier (ROVE) [11] is a formal verification framework which detects incorrect patterns as well as lack of information prevalent among the underlying rule base. At the back end, ROVE makes use of WAVE for the verification process. Based on the concepts discussed in the previous chapter, WAVE has been designed to verify a user specified property in the context of an interactive, web-based database application. Hence, the ROVE framework has been modeled along the template of a web application. The iRODS rule base is stored in a set of iRODS Rule Base (IRB) files which are read by ROVE along with a user specified property. ROVE then invokes the WAVE verifier to prove or disprove the user-specified property. This verification framework is most useful in determining whether changes to the rule bases would affect desired behavior or violate existing dependency constraints.

A basic prototype of the ROVE framework has been designed and implemented previously [11]. The focus of this thesis is the extensions implemented for ROVE which is discussed in the following chapter. Before describing the extensions, it is helpful to understand the semantics used by ROVE.

5.1 Architecture

The main components in the ROVE framework perform completely different functionalities and are shown in Figure 5.1. One component is responsible for the graphical user interface required in order to get the user input. This front-end interface prompts the user for specification of the rule base, property to verify and other options. The other component performs the verification process at the back end. The inputs received from the front-end interface are passed on to this component which translates the received inputs and performs the actual verification.

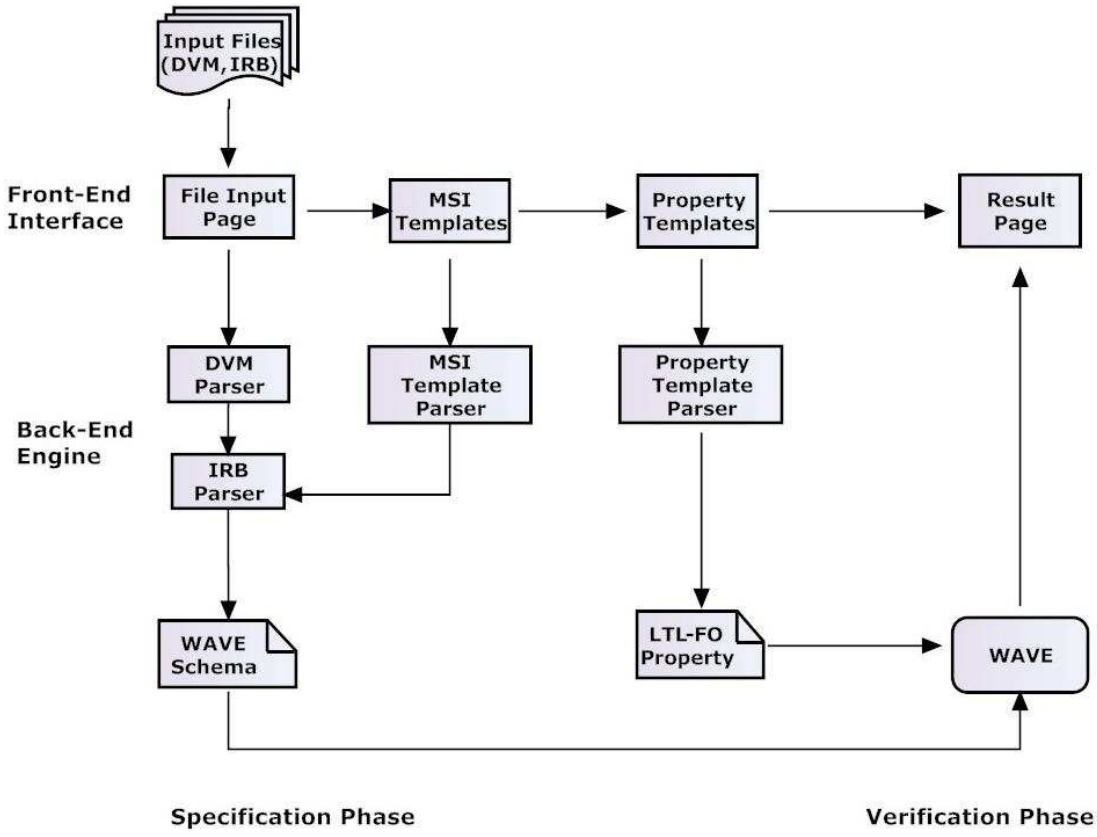

Figure 5.1: ROVE Architecture

The complete verification process performed by ROVE consists of two distinct phases corresponding to the functionalities performed by the above-mentioned two main components of the ROVE architecture [11].

5.2 User Input Phase

The first phase of the verification process is the user input phase or the specification phase. This is divided into two sub-phases that are described below:

5.2.1 File Input Interface

Initial information required for the verification phase is entered by the user through the front-end interface. Through a series of web pages which are part of the front-end interface, the user is first instructed to enter information pertaining to the rule base, session variable mappings and rule definitions. Next, the user is presented with templates to alter or change the existing micro-service definitions. Once the user inputs are received, they are forwarded to the parser which translates the inputs into an output file in the format of the WAVE web page schema.

There are primarily two kinds of files that are used by the ROVE framework. The iRODS Rule Base (IRB) files define the underlying comprehensive rule base against which the user-specified property is verified. The user can input a set of IRB files. Each of these files contains iRODS rule definitions. When the user enters multiple IRB files, the rules in each of them are combined and formed into a single 'master' rule base. The other kind of files that ROVE uses are the Data Variable Mapping or DVM files. As the name suggests, these files

provide a mapping between all the session attributes and their internal names. Similar to IRB files, in the case of DVM files too, the user can enter multiple DVM files which are aggregated into a single 'super' DVM file that contains all the mappings. Both the combined rule base and DVM temporary files are consumed by the back-end parsers in the subsequent phase.

5.2.2 Template Interface

The other part of the specification phase is the part where the user is presented with templates either to modify existing micro-service definitions or to enter a property for verification. Both these interfaces are discussed next.

Micro-service Templates

In this part, the user is presented with a choice of templates to choose from and then the user is prompted to enter property-pertinent information. As before, these inputs are forwarded to the parser which translates them into an LTL-FO file containing the corresponding formula.

However, the typical iRODS user should not be expected to, write entire rule base files using the extended rule language. Hence, an interface [11] has been provided to ease the process of specification which ultimately provides a richer user experience. The template interface facilitates the user to enter variable dependency information. Because of this, users need not hand-code the micro-service rules in the extended language. On modifying the current micro-service as desired, the entered values are parsed into a rule in the extended language format and appended to the master rule base file. Current micro-services are read from a separate micro-service rule base file (which stores all existing

micro-service definitions) and displayed to the user on the micro-service template page.

Property Templates

In this part, the user is presented with a choice of templates to choose from and then the user is prompted to enter property-pertinent information. As before, these inputs are forwarded to the parser which translates them into a LTL-FO (linear temporal logic first-order) file containing the corresponding formula. The extended LTL-FO syntax was explained previously in Section 3.9.

At the back end, the WAVE verifier is invoked to perform verification on properties written in the LTL-FO format. Property templates are a easy and transparent mechanism through which the user can specify properties easily without knowledge of the extended LTL-FO syntax. Three property templates are available currently. They are described below:

- *Permission Templates* - These templates evaluate the property based on currently held user permissions against the micro-services. The micro-service name and currently held permissions are specified by the user. The property evaluates to true if the micro-service specified is invoked successfully with the given list of permissions.

- *Containment Templates* - Invocation of rules may or may not have resultant side effects. Containment templates provide a means to check rules and effects caused by them. The user is presented with a list of actions. It is important to remember that each action is modeled as a rule in iRODS. (Section 3.5) The user selects an action and is then prompted to select variables in order to check if these variables might get affected by invok-

ing the selected action. If all the variables added by the user are modified by the selected action, then the property evaluates to true.

- *Adequacy Templates* - The successful completion of a micro-service requires a set of variables to be pre-filled with non null values. Adequacy templates supply a method to check if the user-specified variables are enough or adequate for the selected action to execute completely. It must be remembered that actions at the lowest granularity are composed of micro-service calls and each micro-service requires a set of variables to be pre-filled in order to complete the required updates. The interface provided enables the user to enter a list of variables and select an action. The property evaluates to true if the given list of variables are adequate for the successful execution of the selected action.

When the user selects the desired property template, the input received from the template pages is parsed and translated into an LTL-FO formula and is stored in a temporary property file. Now, the WAVE verifier is invoked in order to verify this property. WAVE generates the web page schema from the user specified rule base and performs the verification of the property as described in Section 4.2. Finally, the results of the verification are displayed to the user. If the property evaluates to false, a counterexample is displayed.

5.3 Verification Phase

The second phase is where ROVE performs the actual verification. At the end of the user input phase, the user inputs are translated and ready to be consumed by the verification framework. ROVE uses WAVE at the back end for verification. On a higher level, the translated inputs are read by WAVE and

verified according to WAVE semantics as explained earlier. The final result of the verification is shown to the user.

More specifically, when the user completes a property template entirely, the inputs received are parsed into a LTL-FO formula. This LTL-FO formula is stored in a temporary property file. WAVE translates this to the corresponding web page schema with the help of the user specified rule base and ultimately verifies the LTL-FO property formula. The results of the verification are displayed to the user. If the property evaluates to false, a counterexample generated during the verification process is also displayed. Finally, the user is presented with two options - either to verify a new property using the same rule base or return to the start page to verify an entirely new rule base.

The next two chapters describe the extensions made to the ROVE framework, bugs resolved and the requirements for a comprehensive explanation module.

Chapter 6

ROVE Extensions: Part 1 Debugging and Grammar

In the preceding chapters it has been demonstrated that the iRODS rule base can be modeled as a reactive system described in the form of a web page schema. By doing so, WAVE can be used as the back end verifier to validate user specified properties against the rule base. In the previous implementation of ROVE [11] , various functionalities such as micro-service templates, property templates, etc. were provided to the user. These functionalities are only a subset of the entire set of utilities required for realizing the ultimate goal which is to design and implement an exhaustive and comprehensive verifier for the iRODS framework. This chapter and the next describe the extensions made to ROVE since the last implementation. The extensions provide support for a more complex iRODS rule base and augment the existing capabilities of the verifier, thus making it more sophisticated and robust.

This chapter describes the first part of the extensions made that enable ROVE to handle a richer iRODS rule base file. The second part described in the

next chapter focuses on the problem with the existing implementation of the verification results and the necessity of an explanation module for the ROVE user.

6.1 Comments

Support for comments is a highly desirable feature in any rich programming language. Comments describe what is happening, how it is being done, what parameters mean, which globals are used and which are modified, and any restrictions or bugs. Readability of both the DVM and IRB files in iRODS is highly enhanced with support for comments. Any line that is preceded by a '#' symbol in the iRODS rule-base is treated as a comment. The ROVE parser did not support comments so far. The discovery of the lack of support for comments took longer than the actual implementation. Once discovered, a regular expression was added to the backend ROVE parser. This modification ignored any line that was a comment. For example, in Figure 6.1 a sample IRB file is shown. This is how typical iRODS rule base files appear within the iRODS framework. An example of increasing the readability in the code can be seen when comments are used as in Lines 1 through 9 and beyond too.

6.2 Micro-service names

Towards a later stage in development it was discovered that the previous implementation of ROVE [11] required that all iRODS micro-services follow a strict nomenclature. The names of the micro-services needed to be of the form 'msi*' (For example, 'msiCreateUser'). However, the iRODS syntax for rule def-

initions within rule-bases does not impose any restriction of this form. Hence, the parser had to be modified to accept all valid strings as micro-service names.

```
1  # iRODS Rule Base
2  # Each rule consists of four parts separated by  |
3  # The four parts are: name, conditions, function calls and recovery.
4  # The calls and recoveries can be multiple ones, separated by ##.
5  # For each rule, the number of recovery calls should match the calls;
6  # for example, if the 2nd call fails, the 2nd recover call is made.
7  #
8  ##
9  # Test Rules
10 msitest(*A,*B,*E)||msitest1(*A,*B,*C)##msitest2(*C,*D,*E)##msitest3(*A,*D,*E)
11 printHello||print_hello||nop
12 #
13 ##
14 # These are sys admin rules for creating and deleting users
15 #
16 acCreateUser||msiCreateUser##acCreateDefaultCollections##msiCommit|
   msiRollback##nop
17 acVacuum(*arg1)||delayExec(*arg1,msiVacuum,nop)|nop
18 acCreateDefaultCollections||acCreateUserZoneCollections|nop
19 acCreateUserZoneCollections||acCreateCollByAdmin(/$rodsZoneProxy/home,
   $otherUserName)##acDeleteCollByAdmin(/$rodsZoneProxy/trash/home,
   $otherUserName)|nop##nop
20 #acCreateCollByAdmin||msiCreateCollByAdmin($ARG[0],$ARG[1])|nop
21 acCreateCollByAdmin(*parColl, *childColl)||msiCreateCollByAdmin(*parColl,
   *childColl)|nop
22
23 # ----------------------------------------------------------------------
```

Figure 6.1: A sample IRB file

6.3 Call by Reference

To provide for a more powerful programming experience, like other programming languages, iRODS rule syntax also allows call by reference. The iRODS rule definition syntax allows arguments to be passed as 'call by reference' in between micro-services as references. An example demonstrating the

call by reference scheme can be seen on analyzing Figure 6.1. Lines 10, 17 and 21 in the code of the sample IRB file as shown in the figure use call by reference. The best example is Lines 20 and 21 as shown in Figure 6.2. Line 20 contains a previous definition for the action 'acCreateCollByAdmin' which does not use call by reference in its arguments and is hence commented out. Line 21 on the other hand states:

```
20  #acCreateCollByAdmin||msiCreateCollByAdmin($ARG[0],$ARG[1])|nop
21  acCreateCollByAdmin(*parColl, *childColl)||msiCreateCollByAdmin(*parColl,
    *childColl)|nop
```

Figure 6.2: Snippet from the sample IRB file

which is the definition for the action 'acCreateCollByAdmin' using call by reference. The arguments 'parColl' and 'childColl' are passed as references to the micro-service 'msiCreateCollByAdmin'. The ROVE infrastructure was modified to incorporate support for call by reference.

6.4 Cut and Fail

The two keywords 'cut' and 'fail' are part of a special group of workflow services in iRODS. The functionalities of 'cut' and 'fail' keywords are somewhat similar and related. Both keywords are permitted by the iRODS rule syntax to be used within the rule body. Specifically, the 'cut' and 'fail' keywords can be placed in the section of the rule body that contains function calls and micro-service invocations. The following sections elucidate the process of backtracking and the functionalities of the 'cut' and 'fail' keywords.

6.4.1 Backtracking

Rule execution in iRODS follows the PROLOG semantics to a certain extent. When a rule fails, alternative definitions of the rule are retried. For example, consider the following set of rules from an iRODS rule base file:

Rule 1: `acA||acB1##acB2##acB3|nop##nop##nop`

Rule 2: `acA||acC1##acC2|nop##nop`

The rules above present two possible definitions for action 'acA'. Assume acB1, acB2, acB3, acC1 and acC2 are also actions. For sake of simplicity and future references, the rules have been named as 'Rule 1' and 'Rule 2' respectively. Now assume action 'acA' needs to be invoked. The rule definitions are consulted and Rule 1 and Rule 2 are found to match action 'acA'. Rule 1 gets invoked and according to the rule body, action 'acB1' gets invoked which completes successfully. Action 'acB2' is now invoked and assume that it fails for some reason. If no further definition for action 'acA' is available in the rule base, iRODS 'backtracks' to the previous action (in this case 'acB1') and retries it or consults rule base for an alternate definition for 'acB1'. This backtracking is similar to backtracking observed in rule based languages such as PROLOG. In the case where an alternative definition for action 'acA' is found (such as above in Rule 2), iRODS consults the alternative rule (Rule 2) and proceeds as usual. It is vital to understand the concept of backtracking in order to study the semantics of the 'cut' and 'fail' keywords. The following section demonstrates the 'cut' and 'fail' operations.

6.4.2 Cut operation

Cut and fail are primarily used to selectively turn off backtracking. The cut effectively freezes all the decisions made so far in the current rule. That is, if required to backtrack, it will automatically terminate without trying other alternatives. In other words, when the cut is encountered, it re-routes backtracking. It short-circuits backtracking in the actions to its left in the rule body. Consider the same rules as before, but with a slight modification as shown below:

Rule 1: `acA||acB1##cut##acB2##acB3|nop##nop##nop`

Rule 2: `acA||acC1##acC2|nop##nop`

Notice that in Rule 1, the keyword 'cut' has been added. As before, assume action 'acA' needs to be invoked. Rule 1 gets invoked and according to the rule body, action 'acB1' gets invoked which completes successfully. As mentioned before, 'cut' is ignored for now and the next available action from the workflow chain is invoked. In this case, the next available action in the workflow chain is action 'acB2' - which is invoked. Assume that action 'acB2' fails or completes unsuccessfully for some reason. Now, iRODS starts the process of backtracking as mentioned in the previous section. It backtracks through the workflow chain to the previous action. In this case, it encounters a 'cut'. Due to this, it fails action 'acA' entirely without trying alternative definitions for action 'acB1'. This is exactly how the cut operation functions. The current action is terminated and no further retries or recoveries are possible.

Performance is the main reason to use the cut. It can sometimes also have an effect on code readability and maintainability. The cut is analogous to the 'goto' statement seen in many programming languages. It is used when it would be desirable to force a rule to fail in a certain situation, and no further lookups are required.

The 'cut' keyword is modeled as any other micro-service/action with a slight distinction. During the parsing phase, when a 'cut' keyword is encountered, it is simply ignored and the parser moves on to the next token. However, the 'cut' is stored as a token like the other action/micro-services in the rule body. During the WAVE page construction phase, when a 'cut' is detected, the target rules for the 'cut' page specify a return to the action name that contains the 'cut' keyword in its rule body. This is done while constructing the WAVE page objects by keeping track of the appropriate page status and the top of stack information that is annotated with each rule. The truth value 'false' is assigned to the target rule thus signifying that the property evaluates to a false. Since no further backtracking is required, the action simply evaluates to a false.

6.4.3 Fail operation

The fail operation is slightly different from the cut operation in that further recoveries and retires are possible. The rule fails entirely on encountering fail. This might seem similar to the cut operation, but the difference is that although the rule is forced to fail entirely, further retries and recoveries are possible. For example, consider the same rules as before with the keyword 'cut' replaced by 'fail', as shown below:

Rule 1: `acA||acB1##fail##acB2##acB3|nop##nop##nop`

Rule 2: `acA||acC1##acC2|nop##nop`

As before, assume action 'acA' needs to be invoked. Rule 1 gets invoked and according to the rule body, action 'acB1' gets invoked which completes successfully. Now, 'fail' is encountered. The rule evaluates to false and backtracking starts. Action 'acB1' is retried and if it does not complete successfully, the appropriate recovery action is invoked. The fail operation allows for a negation-

like operator. In other words, a typical use of fail is in defining the negation of a rule.

The modeling and implementation of the 'fail' keyword is similar to the 'cut' keyword with the difference in the WAVE page construction phase. As before, during the parsing phase, when a 'fail' keyword is encountered, it is simply ignored and stored as a token like the other action/micro-services in the rule body. During the WAVE page construction phase, when a 'fail' is detected, the target rules for the 'fail' page specify a return to the action/micro-service name that precedes the 'fail' keyword in the rule body. Similar to the implementation of the 'cut', the target rules are specified while constructing the WAVE page objects by keeping track of the appropriate page status and the top of stack information that is annotated with each rule.

Chapter 7

ROVE Extensions: Part 2

Explanation Module

The existing version of ROVE [11] has concentrated primarily on the application of the specification and verification modules from the WAVE framework to the iRODS domain. In section 4.1, the architecture of the WAVE framework was described in detail. One of the modules in the WAVE architecture was the explanation module. As mentioned before, the primary responsibility of the explanation module is to present the results of the verification to the ROVE user in an understandable and user-friendly way. When the user-specified property evaluates to false, a counterexample is generated by WAVE. The generated counterexample (which is actually a directed graph) shows the exact sequence of user inputs and corresponding evolution and point of failure of the web application.

The ROVE user is unaware of the fact that at the back end all inputs received from the user are parsed into WAVE web page schemas and forwarded to the verification module of WAVE which performs the actual verification pro-

cess. It would be highly desirable to maintain this degree of transparency till the very end of the verification process. Hence, when the results of the verification (i.e., the generated counterexample graph) are presented to the user, they must be displayed in a format that the ROVE user understands. This chapter focuses mainly on the above mentioned problem and describes the requirements of a comprehensive module by examples from the current implementation.

In order to understand the examples presented in the later sections, it is helpful to study the algorithms used for translating the ROVE inputs into WAVE web page schema [11]. The web page schema further translates into an intermediate graph representation. The algorithm used for this purpose is described as well. A WAVE generated counterexample is then described. Finally, the last section outlines a desired graphical representation of the generated counterexample that can be understood by the ROVE user.

7.1 Translation from Normal Form to WAVE

Translating the iRODS rules into a WAVE web application is not a simple process. This section attempts at breaking down the complex process involved in the translation. The previous implementation of ROVE [11] has implemented this translation. Section 5.2 details the user input phase of ROVE. An important step in the user input phase is where the user-specified IRB files are concatenated into single rule base file that defines the entire iRODS rule base. On a higher level, as a first step into the translation process, this concatenated rule base is treated as a directed acyclic graph where each node represents an action, a micro-service or an effect. These nodes are later transformed into WAVE pages conforming to the WAVE web page schema as described in Section 4.2. More specifically, there are two parsing phases that accomplish these tasks. The

first phase parses the input specification rules and creates an intermediate representation and the second phase takes the intermediate representation and generates the corresponding WAVE web page schema. These steps are formalized through algorithms implemented in [11] which are described below.

7.1.1 Phase 1: Intermediate Form Generation

The main concept here is that each rule for an action, micro-service or an effect is modeled as a directed acyclic graph. This is similar to modeling a finite state machine with user inputs and corresponding state transitions (as defined in the iRODS rules). The CRB (Concatenated Rule Base) parser is initially invoked to parse the concatenated rule bases and construct a graph containing 'node' objects. The directed acyclic graph that is generated as the intermediate form at the end of this phase has two kinds of nodes:

- *Non-leaf nodes*: A node of this kind in the directed graph can symbolize an operation which could be an action, micro-service or effect invocation. These nodes can expand into other child nodes that correspond to actions, micro-services or effects. Each non-leaf node would maintain a list of its child nodes.
- *Leaf nodes*: A node can be a leaf. Such a leaf node would have no child nodes and would hence correspond to an effect because having no children implies that the task does not expand into sub-tasks.

Formally, the algorithm is shown in Figure 7.1.

Ultimately, when the algorithm completes, the result is a set of disjoint, directed, acyclic graphs. Each node in these graphs represents an action, micro-service (non-leaf nodes) or in the case of leaf nodes - an effect. These disjoint

graphs are then unified into a single directed acyclic graph that is rooted at a particular node. This root node is a newly created node with all the macro-level actions as its children. Macro-level action nodes correspond to the starting nodes in each of the disjoint, directed acyclic graphs that were created as a result of executing the algorithm shown in Figure 7.1.

BEGIN

1) Initially, create an empty hash table to store all the nodes in the graph.

2) Read the concatenated rule base file and parse each iRODS rule in the file in sequential order. Maintain a pointer called '*currentNode*'.

3) For each rule:

(a) Create a new node depending on the type of the task. If it is an action, then create '*acNode*' or '*msiNode*' for a micro-service. Mark this node as the *currentNode*.

(b) Search the hash table using the rule name as a key to see if there already exists a node by the same name.

i. If a node with the same name already exists, then replace it with the new node created in step (a);

ii. If a node with the same name does not exist, insert the newly created node into the hash table.

(c) If the current rule defines an action, traverse the workflow chain and recovery sections of the rule. For each action or micro-service name appearing in these sections, search the hash table to see if there already exists a node with the same name.

i. If such a node already exists, access this node;

ii. If no such node exists, create a new node using the name of the action or micro-service and insert it into the hash table.

iii. Add the node created in step c (ii) above as a child or recovery node to '*currentNode*'.

(d) If the current rule defines a micro-service, traverse the rule body. For each effect, create an '*effectNode*' and add it to '*currentNode*' as a child.

(e) Proceed to the next rule.

END

Figure 7.1: Intermediate graph generation algorithm

7.1.2 Phase 2: WAVE Web Page Schema Generation

At the end of the first phase, an intermediate representation in the form of a rooted directed acyclic graph is generated. ROVE uses the WAVE verifier at the back end. Hence, in order to use the verification functionalities of WAVE, the intermediate form generated at the end of the first phase needs to be translated into a WAVE web page schema.

The second phase is responsible for the actual translation. It takes the intermediate graph representation as input and produces the corresponding WAVE web page schema. Technically, the second phase generates 'WAVEPage' objects from the nodes of the intermediate graph. The intermediate graph is traversed recursively, starting at the root which contains all the macro-level action nodes as it children. At each step of the traversal, the buildPage() method is called to annotate the pages with input rules, state rules, target pages, and target rules. The algorithm [11] shown in Figure 7.2 formalizes these steps.

When the algorithm terminates, various 'WAVEPage' objects are created and the generated schema is printed out to a file. Having outlined the translation algorithms above, the following sections outline the problem with the existing display of the verification results to the ROVE user and then focus on the requirements for an explanation module.

BEGIN
1) Start a depth-first traversal of the graph at the root node. Generate a '*HomePage*' object for the root node.

2) Generate a web page for each visited node depending on the type of the node. For example, action nodes and micro-service nodes would generate '*StandardPage*' objects, while effect nodes would generate '*EffectPage*' objects.

3) For action nodes and micro-service nodes:

(a) For each child node, invoke the '*buildPage()*' method and pass the current page as a parent page. The method should return the '*WAVEPage*' object that represents this particular child. Add this object as a 'child page' to the current page.

(b) Repeat step 3 (a) for recovery nodes, if there are any.

(c) Annotate state rules and target rules for the current page based upon the parent page and the child pages as follows:

i. The parent page is visited when all child pages complete successfully, or if one of them fails and all recovery actions are completed.

ii. A child page is visited only when all of the previous child pages have successfully completed.

iii. A recovery page is visited only when the corresponding child page fails and needs to be rolled back.

4) For each effect node, create a new page and generate the state rules. An effect page is considered to be a child page of the micro-service that contains it.
END

Figure 7.2: WAVE Web Page Schema Generation Algorithm

7.2 Verification Results Page

When the user completes a property template entirely, as described previously, the inputs received are parsed into a LTL-FO formula which is translated by WAVE to the corresponding web page schema and verified at the back end. The results of the property verification are displayed to the user. If the property evaluates to false, a counterexample generated during the verification process is also displayed. Finally, the user is presented with two options - either to verify a new property using the same rule base or return to the start page to verify an entirely new rule base. A sample screenshot of the verification results page is shown in Figure 7.3.

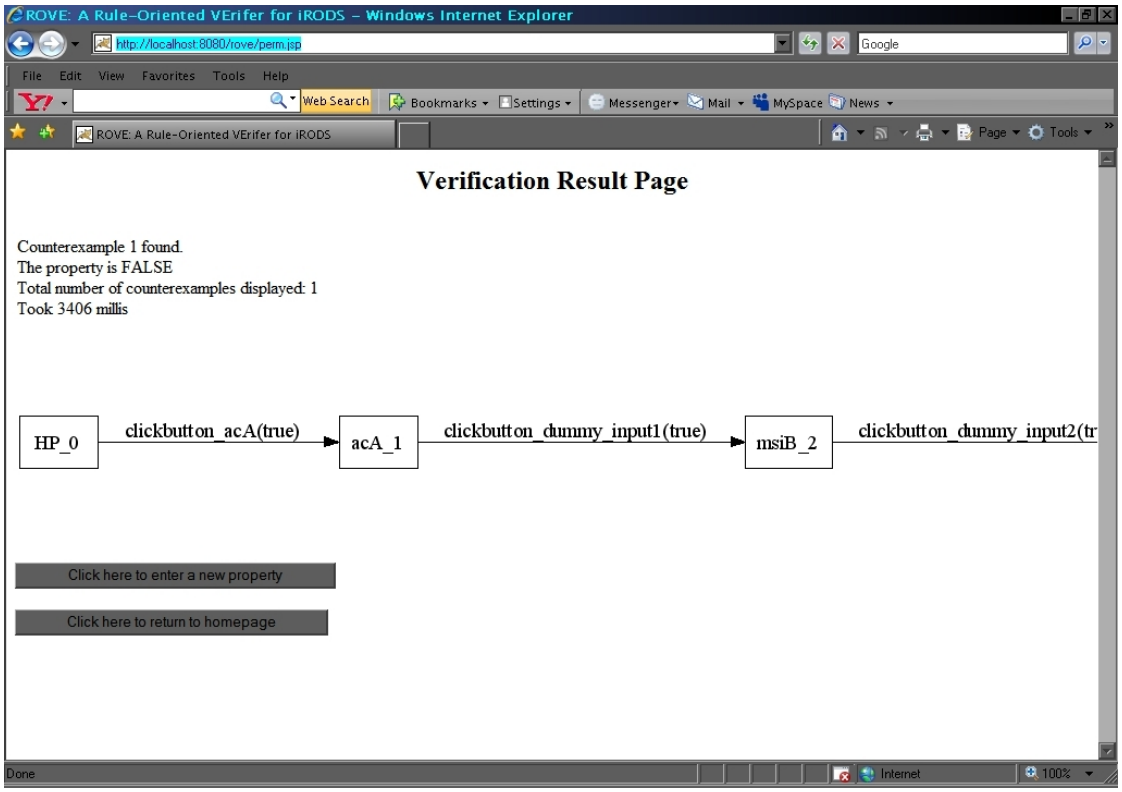

Figure 7.3: Verification Result Page

The result page shown in Figure 7.3 displays results of verification of a user specified property. It is evident from the figure, the property has evaluated to false and hence a counterexample has been displayed. The counterexample is displayed according to WAVE semantics. It is a graphical representation of the traversal of a subset of the nodes in the intermediate graph created in phase 1 (as described in subsection 7.1.1). The ROVE user is familiar with iRODS semantics and rule definitions and is completely unaware of the invocation of WAVE as the verifier at the back end. Hence, it is vital to present the results of the verification - more specifically, the counterexample, in a jargon that will be understood by the ROVE user. The next section illustrates the creation of a counterexample when an example property is verified against a sample rule base.

7.3 A ROVE Verification Result Counterexample

This section presents an example demonstrating a user specified property that evaluates to false against a simple rule base. The main purpose is to point out the complexity of the counterexample generated in comparison to the simplicity of the sample rule base read in. By doing so, it is clear that there is an urgent requirement to modify the presentation of the verification results and devise an algorithm that generates a graphical counterexample that is easily understood by the ROVE user.

The ROVE framework accepts from the user a set of iRODS Rule Base (IRB) and Data Variable Mapping (DVM) files as described in Section 5.2. As an example, consider that the sample IRB and DVM files shown in Figure 7.4 are fed into ROVE as an input. While running the ROVE application in the browser, once the IRB and DVM files are accepted, the user navigates away

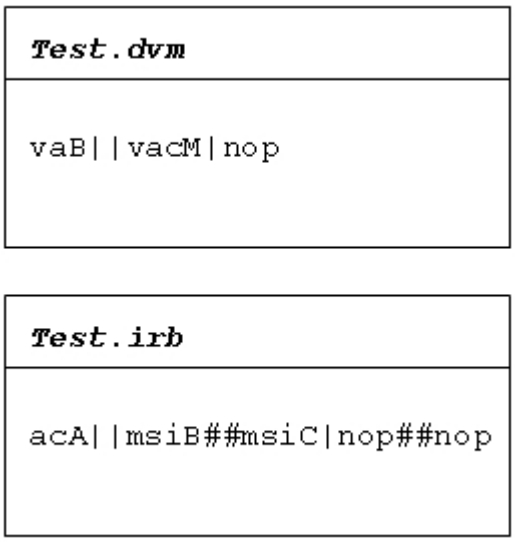

Figure 7.4: Sample IRB and DVM Files

from the input page and reaches the page where modifications to an existing action/micro-service (as defined in the input IRB files) can be made.

The IRB and DVM files in Figure 7.4 are chosen to be relatively simple. The DVM file contains just one mapping and the IRB file contains a rule that defines one action. Action 'acA' is defined by the rule in the IRB file. Invocation of action 'acA' results in invoking micro-services 'msiB' and 'msiC' respectively. In order to keep it simple, recovery operations are 'nop's (no recovery routine defined). The user is presented with the edit micro-service page, where the definitions of micro-services 'msiB' and 'msiC' can be changed. Figure 7.5 illustrates the edit micro-service page for clarity. Assume that for the given IRB and DVM files, the user makes the following changes to the micro-services.

- For micro-service 'msiB' the user adds the condition '$vaB==abc'
- For micro-service 'msiC' the user adds a dummy effect '$vaB<−$vaB'

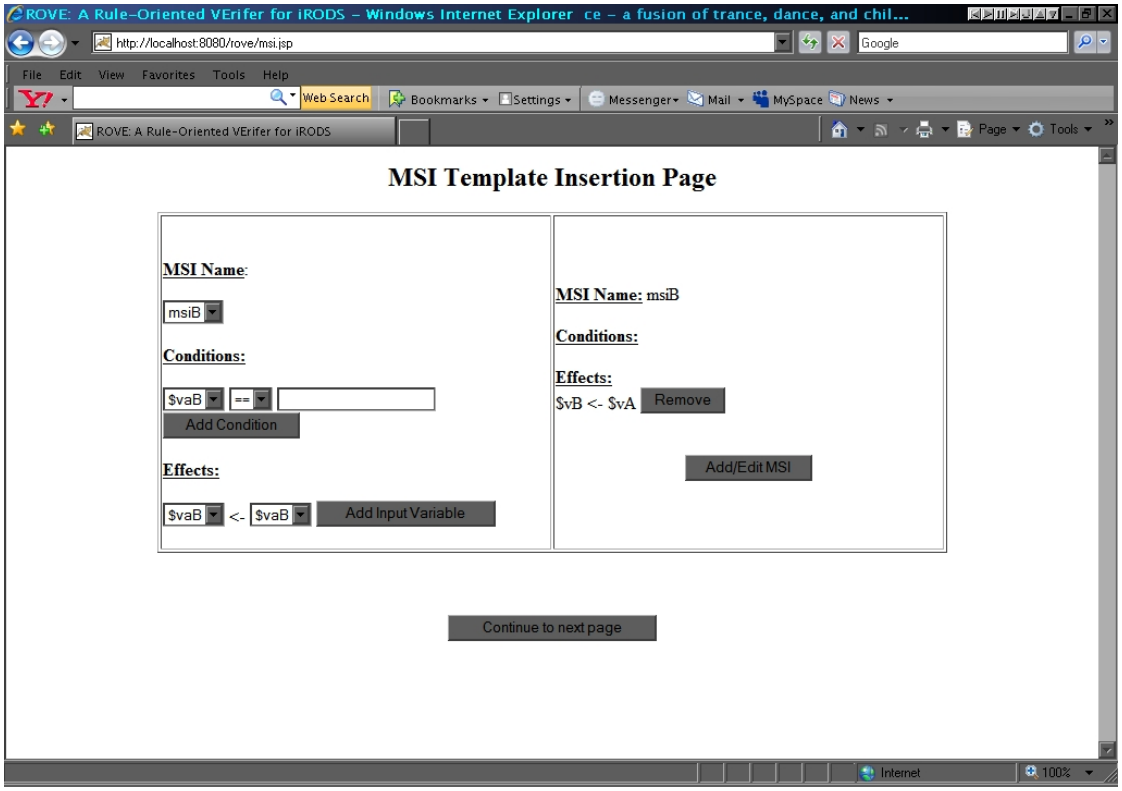

Figure 7.5: Micro-service Edit Page

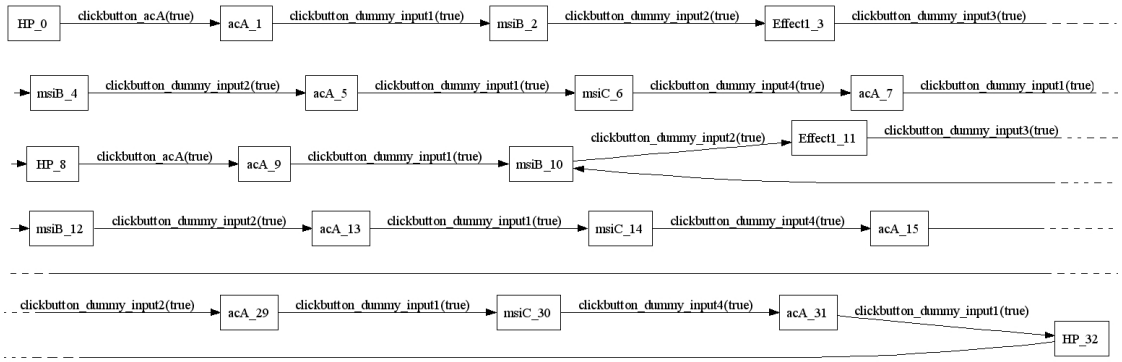

Figure 7.6: Generated Counterexample

To reiterate, the rule definitions, actions, conditions and effects have been kept as simple as possible to illustrate the complexity of the counter example generated with an uncomplicated environment. After making these changes, the user is prompted with the property template selection page listing the three available property templates as described in subsection 5.2.2. Assume that the user selects the 'Permission Template' to verify a property pertinent to permissions held. On the add permissions page, the user enters a dummy permission '$vaB==v' and checks to see if given this permission, micro-service 'msiC' can be invoked. The result of the property verification is false and a counterexample is displayed on the results page. The counterexample is a graph consisting of a multitude of nodes showing an execution sequence composed of micro-services 'msiB' and 'msiC' along with action 'acA'. It is shown in Figure 7.6. (To accommodate the entire graph on one page, it has been split-up line wise.)

From the context of WAVE, this counterexample is complete and sufficient. However, for a ROVE user, the counterexample is not easy to understand. For example, the transitions between the nodes of the graph are labeled as 'clickbutton_' which actually refers to transition rules within the WAVE web page schema - something the ROVE user is unaware of. Hence, it is important to

depict the counterexample in a manner that the ROVE or iRODS user can understand. The next section presents an example of such a graphical, user-friendly counterexample.

7.4 An Ideal ROVE User-Friendly Counterexample

The ROVE user is familiar with iRODS rules and the execution environment. The iRODS framework architecture and environment have been described in detail in Chapter 3. The ultimate design goal for iRODS is to define management policies and to automate the application of these policies for a multitude of data management services. Using the paradigm of rule-oriented programming, these management policies are mapped onto rules that control the execution and customization of all data management operations in an easy and declarative fashion. The complete power of iRODS can be exploited only when the customizability is maintained throughout the graphical user interface that is presented to the user. Hence, the counterexample that is produced at the end of the verification process must contain the same actions, micro-services or effects that the user started with at the beginning of the verification process. The existing format of the counterexamples generated has been presented via an example in the previous section. The problem is now to translate the WAVE produced counterexample into a format that the ROVE user can assimilate.

Firstly, let us take a closer look at the existing method of representing the counterexample. The counterexample is presented as a directed graph (as in Section 7.3). The directed graph demonstrates the transitions in between various stages of the verification process. However, the generation of the counterexample graph is a part of WAVE and hence, the nomenclature used in the generated graph is in the context of WAVE. Hence, the main problem is in translating this

generated graph into a more meaningful and intuitive form that the ROVE user can comprehend. For example, instead of having 'web pages' the counterexample should have action/micro-service names and it should be able to capture the input-output relations existing therein.

The next section outlines the architecture, explains the functions and justifies the completeness of the representation generated by the new explanation module.

7.5 Explanation Module: Architecture

The main goal of the explanation module is to translate the counterexample generated by WAVE[3] into a more meaningful and lucid representation. The explanation module accomplishes this translation in two distinct phases. The first phase involves the generation of the counterexample through WAVE in a text-based format. The text-based counterexample generated by WAVE consists of significantly scattered but relevant data that is logged as WAVE generates the counterexample for the false property. When the user reaches the 'Verification Results' page within the ROVE interface, at the back-end, a specification file 'temp_spec.txt' is generated. This file contains a text-based representation of the ROVE rule-base as WAVE objects. The algorithm for generating this text file has been outlined in Figure 7.2. A snapshot of the file is shown below (Figure 7.7). This file serves as a 'dictionary' for looking up the input/output relations between the various actions and micro-services defined in the rule-base.

```
1   Input:
2       clickbutton_acA(x0);
3       .
4   Web Page Schema Set:
5       HP; msiC; msiB; acA
6   Home Page: HP
7   Error Page: EP
8
9   Page: HP(HP)
10  Input:
11      clickbutton_acA(x0)
12  Input Rules:
13      Options@clickbutton_acA(x0) : = x0 = "true"
14  State Rules:
15      not Page_Status(x, y) : = Page_Status(x, y);
16      not TopOfStack(x, y) : = TopOfStack(x, y);
17                  .
18                  .
19      Page_Status("HP", "unvisited") : = clickbutton_acA("true");
20      Page_Status("acA", "unvisited") : = clickbutton_acA("true");
21      Page_Status("msiB", "unvisited") : = clickbutton_acA("true");
22                  .
23                  .
24  Target Webpages:
25      acA
26  Target Rules:
27      acA : = clickbutton_acA("true")
28
29
30  Page: msiC(msiC)
31  Input:
32      clickbutton_dummy_input3(x)
```

Figure 7.7: Specification File 'temp_spec.txt' generated by WAVE [3]

7.5.1 Temporary output

At the end of the verification, the results are logged into another text file called 'temp_output.txt'. This file contains the actual result of the verification and additional temporary data logged during the process. A snapshot of the file can be seen in Figures 7.8 and 7.9. The important essence from this file to use is the counterexample data.

```
**** Input variables:
{clickbutton_dummy_input20=X3, clickbutton_acA0=X1, clickbutton_dummy_input30=X2, clickbutton_dummy_input10=X4

**** Web pages and input variables:
{msiC=[1, X2, Button], msiB=[1, X3, Button], acA=[1, X4, Button], HP=[1, X1, Button]}

**** Relations and attributes:
{clickbutton_dummy_input3=[x], clickbutton_dummy_input2=[x], clickbutton_dummy_input1=[x], Pass=[x], Page_Stat

Edges are: From X1   To:  true    Edge type: Dot line
                .
**** Start Processing Property ****
Original formula: ! ([] (! (<> (Page_Status("msiC" "visiting")))))
Formula after normalization (if any): ! ([] (! (<> (Page_Status("msiC" "visiting")))))
**** End Processing Property ****
All possible combinations of input constants: [{}]

Input constants selection: {} 1 out of 1
    Universal variables selection: {} 1 out of 1
    Propositional LTL formula obtained: ! ([] (! (<> (p1))))
    p1: Page_Status("msiC" "visiting")
    All possible tuples for core: []
        Core: 1 out of 1
        Core database:
        Current page: HP
            All valid inputs: [{F1=0, X1=true}, {}]
            Current input: {F1=0, X1=true} 1 out of 2
                Extension: 1 out of 1
                Extension database:
                    Target page: acA
        Current page: acA
            All valid inputs: [{X4=true, F4=0}, {}]
```

Figure 7.8: WAVE[3] Verification Output 'temp_output.txt'

The counterexample generated by WAVE [3] is written at the end of the text file ('temp_output.txt'). This information is extracted and written to a file 'output.txt' in the format showin in Figure 7.10.

This corresponds to a 'run' of events that falsifies the user-specified property. As evident from the figure above, it is clear that the format of the generated counterexample shown in Figure 7.10 is not self-explanatory. This data is now

```
                Extension database:
                    Target page: msiC
            Current page: msiC
                All valid inputs: [{F2=0, X2=true}, {}]
                Current input: {F2=0, X2=true} 1 out of 2
                    Extension: 1 out of 1
                    Extension database:
                        Target page: acA
            Current page: acA
                All valid inputs: [{X4=true, F4=0}, {}]
                Current input: {X4=true, F4=0} 1 out of 2
                    Extension: 1 out of 1
                    Extension database:
                        Target page: HP
            Current page: HP
                All valid inputs: [{F1=0, X1=true}, {}]
                Current input: {F1=0, X1=true} 1 out of 2
                    Extension: 1 out of 1
                    Extension database:
                        Target page: acA
                Current input: {} 2 out of 2
                    Extension: 1 out of 1
                    Extension database:
                        Target page: HP
    The property is FALSE
    *** Counterexample found, quitting... ***
    HP<state0_-1(final)(initial)>t
    [HP{F1=0, X1=true}, acA{X4=true, F4=0}, msiB{F3=0, X3=true}, acA{X4=true, F4=0}, msiC{F2=0, X2=true},
    acA{X4=true, F4=0}, HP{}, HP{F1=0, X1=true}, acA{X4=true, F4=0}, msiB{F3=0, X3=true}, acA{X4=true, F4=0},
     msiC{F2=0, X2=true}, acA{X4=true, F4=0}]
            Core database:
    Took 297 millis
```

Figure 7.9: WAVE[3] Verification Output 'temp_output.txt'

broken down in order to generate a cleaner representation. Each of the tuples in the format above corresponds to a single node in a directed graph with the specified inputs. It must be noted that WAVE stops as soon as the first counterexample is generated. It is not necessary that the generated counterexample is minimal. There might be optimal runs of the 'web pages' that produce a smaller and more optimal counterexample, but that is out of the scope of this project.

```
    *** Counterexample found, quitting... ***
    HP<state0_-1(final)(initial)>t
    [HP{F1=0, X1=true}, acA{X4=true, F4=0}, msiB{F3=0, X3=true}, acA{X4=true, F4=0}, msiC{F2=0, X2=true},
    acA{X4=true, F4=0}, HP{}, HP{F1=0, X1=true}, acA{X4=true, F4=0}, msiB{F3=0, X3=true}, acA{X4=true, F4=0},
     msiC{F2=0, X2=true}, acA{X4=true, F4=0}]
            Core database:
    Took 297 millis
```

Figure 7.10: Text representation of the counterexample

The final result of the explanation module is the generation of a formatted, text-based 'explanation output' file (output.txt) that contains the translated counterexample which is completely in the context of ROVE and iRODS. Auxiliary data is also written out to enable readability for the end user.

An example of the final output file generated for a simple counterexample is shown in Figure 7.11.

The following subsections illustrate how the output file is generated.

7.5.2 Generation of the output file

To reiterate, the format of the counterexample generated by WAVE [3] is shown in Figure 7.10.

This corresponds to a 'run' of events that falsifies the user-specified property. To generate a readable result, a translation process is performed. Each of the tuples in the format above corresponds to a single node in a directed graph with the specified inputs. The following subsections break up each individual piece of data and demonstrate the translation of each of them into the corresponding iRODS components.

WAVE Pages

The first piece of information contained in each tuple in the generated counterexample is the WAVE page corresponding to the current action or micro-service. The input parameters for each of these action/micro-service pages are explained in the next section. Extracting only the name of the WAVE page is not enough as this information needs to be 'reverse-mapped' to the corre-

```
1  Counterexample successfully generated!
2  Please follow the steps below:
3
4  Step 1
5  =============
6  Start of rule-base. First action to be invoked: acA
7  Invoked: Action acA
8
9  Step 2
10 =============
11 Current Action/micro-service: acA,
12 Action unchanged
13       Possible Target Actions/MSIs: msiB msiC
14 Invoked: Micro-service msiB
15
16 Step 3
17 =============
18 Current Action/micro-service: msiB,
19       Inputs:
20 User input for micro-service received and used
21 User input:
22 Conditions entered:  $acA   ==  a,
23 Effects entered   :  $acA  <-  $acB,
24       Possible Target Actions/MSIs: acA
25 Invoked: Action acA
26
27 Step 4
28 =============
29 Current Action/micro-service: acA,
30 Action unchanged
31       Possible Target Actions/MSIs: msiB msiC
32 Invoked: Micro-service msiC
```

Figure 7.11: Output of the explanation module

sponding iRODS action/micro-service. The 'temp_spec.txt' file is consulted to get the precise specification of the current WAVE page. The corresponding iRODS action/micro-service name is extracted and written to the output file. The 'temp_spec.txt' file also gives information about all possible combinations of input constants associated with each WAVE page. To make the explanation module comprehensive, the final output file also contains a list of the iRODS actions/micro-services that can be invoked from the current action/micro-service. This reinforces the validity of the underlying rule-base to the end user.

Inputs

The inputs seen in Figure 7.8 are labeled as 'Fx' or 'Xx' where 'x' is an integer. The 'F' inputs denote a flag; signifying whether the inputs to the corresponding micro-services have been used or not. If set to '1', it denotes that the inputs are not used, if set to '0', it denotes that the inputs are used. On the other hand, if the 'X' inputs are set to 'true', it signifies that the condition associated with the micro-service either holds or there is no condition. In the WAVE terminology, the 'X' inputs correspond to the simulation of a button click on a webpage. In the case of ROVE, they correspond to 'dummy' buttons so that the actions/micro-services can be modeled as web pages to be verified by WAVE. The information of each 'X' input is stored in the 'temp_spec.txt' file. The 'F' and 'X' input variables are important in reconstructing the counterexample during the translation process. Similar to looking up definitions of words in a dictionary, the 'dictionary' file ('temp_spec.txt') is now consulted for decoding each of the pieces in the counterexample data. To capture the input/output dependencies, it is not enough to lookup the 'dictionary' file as it only has information for the WAVE pages.

Conditions and Effects

The input/output conditions for each micro-service and action are obtained from the rule-base. In the case where the user makes a change to an existing micro-service definition by altering a condition or an effect, while on the 'MSI Template Insertion Page' of the ROVE user interface, these must also be captured. Any changes made to the 'MSI Template Insertion Page' are logged appropriately in the background as this information is useful while comparing and checking for the correct conditions and effects associated with each micro-service. The changes made to any micro-service are stored in the 'conds.txt' file within the ROVE package. A sample 'conds.txt' file is as shown in Figure 7.12.

```
1   msiB|||msiB| $acA   ==  first,||
2   msiB| $acA   ==  first,||
3   msiB| $acA   ==  first,| $acA  <-  $acA,|
4   msiB| $acA   ==  first, $acA   ==  second,| $acA  <-  $acA,|
5   msiB| $acA   ==  first, $acA   ==  second,| $acA  <-  $acA,|
6   msiB| $acA   ==  first, $acA   ==  second,| $acA  <-  $acA, $acA  <-  $acB,|
7   msiC|||msiC| $acA   ==  cfirst,||
8   msiC| $acA   ==  cfirst,||
9   msiC| $acA   ==  cfirst,| $acA  <-  $acA,|
10  msiC| $acA   ==  cfirst, $acA   ==  csecond,| $acA  <-  $acA,|
11  msiC| $acA   ==  cfirst, $acA   ==  csecond,| $acA  <-  $acA,|
12  msiC| $acA   ==  cfirst, $acA   ==  csecond,| $acA  <-  $acA, $acA  <-  $acA,|
```

Figure 7.12: Conditions captured by explanation module

The 'conds.txt' file essentially captures all the user inputs on the 'MSI Template Insertion Page' of the ROVE user interface. The explanation module checks the conditions associated with each micro-service/action present in the counterexample that is generated by WAVE. These are written out to the output file so that the user can also view the conditions entered (if any) or existing conditions associated with each micro-service/action.

Final Output

The output file is the final result of the explanation module and it is shown in Figure 7.11. The starting point is always a 'Home Page' or start point. Then, on the basis of data generated by WAVE [3] for the counterexample (Figure 7.10), the explanation module output is written out. The appropriate user-input status is printed - whether a user made any changes to the micro-service definition or not. The conditions and effects are also displayed, if a change has been made. Then, a possible list of micro-services or actions is displayed after consulting the rule-base. The next invoked action/micro-service should be from this list and is defined as the next node while traversing the counterexample graph generated by WAVE. A sample final output file after running the explanation module is shown in Figure 7.11.

To summarize, the interdependencies of all the components of the explanation module can be illustrated in Figure 7.13.

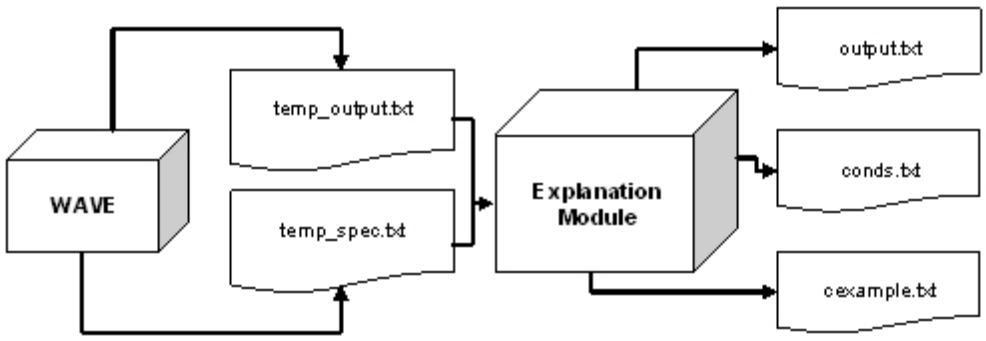

Figure 7.13: Components of the Explanation module

Chapter 8

Conclusion and Future Work

A powerful data grid such as the iRODS framework provides comprehensive mechanisms needed to manage distributed data, the tools that simplify automation of data management processes, and the logical name spaces needed to assemble collections. Additionally, it seeks to fortify significantly important aspects of software engineering such as flexibility and customizability. The user designs, implements and controls data management policies to meet specific demands. At the very core, this is done by utilizing the design philosophies of rule-oriented programming.

The ROVE framework [11] is used as a formal verification framework to verify user specified properties against an iRODS rule base. At the back end, ROVE invokes the Web Application VErifier (WAVE) for carrying out the verification. This thesis has outlined the goals for iRODS and has described in great detail each of the components that are involved in achieving these goals. More specifically, the main focus problem is the establishment of the requirements for a better explanation module for the ROVE user. Extensions to the framework that make it more robust and efficient have also been detailed. However, the

project still involves and presents a significant scope for additional future work that is outlined below.

- **Formalizing the Counterexample Representation**
  This thesis has outlined and described the existing problem with the counterexample representation. An ideal alternative representation has also been proposed. However, this area is still open to further research that can facilitate a more optimal and user-friendly representation. Once a representation has been selected, the developer would need to formalize the algorithms involved.

- **Debugging**
  In this version of ROVE, extensive detail has been laid on minimizing the number of errors as the user transitions through the application. However, with varying complexity of the underlying rule-base, the probability of erroneous behavior is not negligible. Exhaustive testing has always been stressed upon and it should continue with the addition of each new functionality in the framework. This would enable the iRODS framework as a whole to be more efficient, robust and easy to deploy and maintain.

- **Novel Property Templates**
  There are three property templates that are implemented presently in ROVE [11]. There is a significant scope for exploring and designing newer property templates that emulate commonly used properties in iRODS.

- **Extending Micro-service Modeling**
  The existing version of ROVE [11] captures micro-services as input output variable dependencies. The behavior pattern of micro-services can

be further researched. This would enable possibilities for a richer model that could be used to emulate micro-services.

- **Verification of Side-Effects**
  When actions are executed, underlying micro-services are invoked. Invocation of micro-services leads to certain side-effects that can alter the existing state. Verification of the side-effects would enable a higher level of security and robustness.

- **Modeling Complex Conditions**
  ROVE supports the equality and inequality conditions in its current prototype. Further research can be carried out to implement condition expressions containing complex arithmetic operators.

The aspiration is to ultimately incorporate all the above mentioned ideas into the iRODS framework so that its complete power can be exploited.

References

[1] S. Abiteboul, L. Herr, and J. V. den Bussche. Temporal versus first-order logic to query temporal databases. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 49–57, Montreal, Canada, 3–5 June 1996.

[2] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The sdsc storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1998.

[3] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A verifier for interactive, data-driven web applications. In F. Özcan, editor, *SIGMOD Conference*, pages 539–550. ACM, 2005.

[4] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web services. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 71–82, New York, NY, USA, 2004. ACM Press.

[5] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. A system for specification and verification of interactive, data-driven web applications. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 772–774, New York, NY, USA, 2006. ACM Press.

[6] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicat-

ing data-driven web services. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 90–99, New York, NY, USA, 2006. ACM Press.

[7] http://db.ucsd.edu/wave/. Web application verifier project site.

[8] http://irods.sdsc.edu. Irods wiki.

[9] http://www.sdsc.edu/srb/index.php. Storage resource broker.

[10] A. Jagatheesan and A. Rajasekar. Data grid management systems. In ACM, editor, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pages 683–683, pub-ACM:adr, 2003. ACM Press.

[11] R. S. Liu. A verification framework for a rule-oriented data grid management system. In *Thesis (M.S.) University of California San Diego*, 2007.

[12] R. Moore, A. Rajasekar, and M. Wan. Data grids, digital libraries and persistent archives: An integrated approach to publishing, sharing and archiving data, vol. 93, no.3. In *Special Issue of the Proceedings of the IEEE on Grid Computing*, pages 578–588, 2005.

[13] A. Rajasekar, R. Moore, M. Wan, and W. Schroeder. A prototype rule-based distributed data management system. In *HPDC Workshop on Next-Generation Distributed Data Management*, June 20, 2006.

[14] A. Rajasekar, M. Wan, R. Moore, and W. Schroeder. Data grid federation. In *PDPTA*, pages 541–546, 2004.