Original version by Joseph Kiniry, with assistance from Cesare Tinelli and Silvio Ranise beginning on 21 June 2004.

Current editor Joseph Kiniry.

This document describes the provers used by ESC/Java2. It describes the expected general interface (for I/O) and functionality (mathematically, in terms of built-in theories, and functionally, in terms of proof commands) of ESC/Java2 provers, provides a ESC/Java2 Prover API in several forms (as a JML-annotated API, as a WDSL web service API, etc.), provides implementation details for distributed, current, and high-performance prover interfaces.

This is edition $Revision$.

This document is a work in progress. Suggestions and input are always welcome.

The Provers of ESC/Java2

Their Necessary Functionality, Interfaces, and Use
Edition $Revision$, May 2004
This document describes the provers of ESC/Java2 version 2.0a7.

Joseph R. Kiniry <kiniry@acm.org>

Table of Contents

1 Introduction **2**

2 The ESC/Java2 Prover API **3**

2.1 A JML API 3
2.2 A WSDL API 3

3 Prover Communication **4**

3.1 Using Raw Inter-Process Communication (IPC) 4
3.2 Using Java RMI 4
3.3 Using XML-RPC 4
3.4 Using a Java Native Interface (JNI) 4

4 ESC/Java2 Prover Built-in Theories **6**

5 References **7**

6 Copying **8**

Appendix A JML API **9**

Index **15**

This document describes the provers used by ESC/Java2. It describes the expected general interface (for I/O) and functionality (mathematically, in terms of built-in theories, and functionally, in terms of proof commands) of ESC/Java2 provers, provides a ESC/Java2 Prover API in several forms (as a JML-annotated API, as a WDSL web service API, etc.), provides implementation details for distributed, current, and high-performance prover interfaces.

1 Introduction

* Discuss what this document contains.

2 The ESC/Java2 Prover API

2.1 A JML API

Appendix A [JML API], page 9 includes the JML specifications of the core classes of this API.

2.2 A WSDL API

3 Prover Communication

The ESC/Java2 front-end must communicate with one or more prover back-ends in some fashion, as it does not have a “build-in” prover. Whereas the ESC/Java2 front-end is written in Java and thus must run in a Java Virtual Machine process, the back-end provers can be written in any language.

If the back-end is written in Java, then communication between the front- and back-ends can be directly accomplished through (perhaps remote) Java method invocation. As far as we know, no first-order prover has been written in Java, thus this is only a theoretical option.

All existing first-order provers are written in languages other than Java. The languages used in modern provers include Modula-III, C, C++, ML, and OCaml.

Communicating between a Java process and a non-Java subsystem (either a library, DLL, or process), is a non-trivial task. It can be accomplished in any number of ways including native inter-process communication (IPC), XML-based remote procedure calls (XML-RPC), or directly function calls through a Java Native Interface (JNI).

In general, one must use the JDK 1.4 “new I/O” (NIO) interface to obtain the highest-performance inter-subsystem communication. Using NIO, the contents of a buffer can, potentially, reside in native memory outside of the ordinary garbage-collected heap. Manipulation of these buffers, both from within Java and within native code, is extremely fast on many common VMs, operating systems, and architectures.

This chapter discusses these different alternatives, comparing and contrasting them according to many criteria including performance, maintainability, flexibility, simplicity, and robustness.

3.1 Using Raw Inter-Process Communication (IPC)

There are several kinds of inter-process communication (IPC). The most popular variants include signals, anonymous or named pipes, files and file locking, message queues, semaphores, shared memory, memory-mapped files, and sockets. On many operating systems, IPC between local processes is optimized using high-throughput, low-latency mechanisms like shared memory. A decent tutorial on IPC exists at `http://www.ecst.csuchico.edu/~beej/guide/ipc/`.

“Raw” inter-process communication is the means by which SRC ESC/Java communicated with the Simplify theorem prover. The standard input (STDIN) and standard output (STDOUT) streams were used to send bytes between the (Java-based) SRC ESC/Java front-end and the (Modula-III-based) Simplify back-end. These streams are realized with anonymous pipes on most platforms.

While this kind of IPC is very simple to set up and use, it is not high performance. Performance is poor because every byte sent through the stream is typically copied several times before it is received. Additionally, streams must be flushed to ensure data is delivered to the receiver. This is a programmatic inconvenience which contributes to the most frequently witnessed IPC bug—that of mis-synchronization.

3.2 Using Java RMI

3.3 Using XML-RPC

3.4 Using a Java Native Interface (JNI)

The Java Native Interface is is a standard programming interface for writing Java native methods and embedding the Java virtual machine into native applications or vice-versa. The primary goal is binary compatibility of native method libraries across all Java virtual machine implementations on a given platform.

Coupling a Java-based application to a non-Java-based application or library using JNI provides the highest possible performance for inter-system communication in many circumstances. If in a given design cross-subsystem calls (a) are very common (their absolute count is very high), or (b) must have *very* low latency, or (c) must send or receive large amounts of non-referential data, then JNI is an inappropriate solution for such a design.

The JNI is primarily documented in the JDK via the *Java Native Interface Specification*. It is also thoroughly documented in the book *"The Java Native Interface: Programmer's Guide and Specification" by Sheng Liang, Addison-Wesley, 1999*, available via Amazon and other online book retailers. Sun also provides a tutorial, *"Java Native Interface", by Beth Stearns* available via `http://java.sun.com/docs/books/tutorial/native1.1/`.

For the highest-performance inter-subsystem communication using JNI, the JDK 1.4 "new I/O" (NIO) interface should be used. This functionality, new to JDK 1.4, permits native code to access `java.nio` direct buffers.

4 ESC/Java2 Prover Built-in Theories

5 References

6 Copying

Appendix A JML API

The main interface of the prover is specified in the Java interface `ProverInterface`.

```
import java.util.Properties;

interface ProverInterface
{

    /*
     * Variables indicating the state of the prover
     * All assertions using this have been commented for the moment
     */
//     public boolean prover_started;
//     public boolean signature_defined;

  /**
   * Start up the prover.  After the prover is started correctly it
   * should be ready to receive any of the commands embodied by all
   * the other methods of this API.
   *
   * @return a response code.  A response code of {@link
   * ProverResponse#OK} indicates that the prover started successfully
   * and is ready to receive commands.  A response code of {@link
   * ProverResponse#FAIL} indicates that the prover did not start
   * successfully and is not ready to receive commands.  In the latter
   * case, {@link ProverResponse.FAIL.info} can contain additional
   * arbitrary information about the failure.
   */
  /*@ public normal_behavior
    @   ensures \result == ProverResponse.OK ||
    @           \result == ProverResponse.FAIL;
    @*/

  public /*@ non_null @*/ ProverResponse start_prover();

  /**
   * Send arbitrary information to the prover.  Typically this
   * information is not mandatory and is only suggestions or
   * informative information from the front-end.  This data is highly
   * prover-dependent.
   *
   * @param properties the set of property/value pairs to send to the
   * prover.
   * @return a response code.
   */
  /*@ public normal_behavior
//    @   requires prover_started;
    @   ensures \result == ProverResponse.OK ||
    @           \result == ProverResponse.FAIL ||
    @           \result == ProverResponse.SYNTAX_ERROR ||
    @           \result == ProverResponse.PROGRESS_INFORMATION;
    @*/
  public /*@ non_null @*/ ProverResponse set_prover_resource_flags(/*@ non_null @*/ Prop

  /**
   * Send the signature of a new theory to the prover.
   *
   * Note that an empty signature is denoted by an empty {@link
   * Signature} object, <em>not</em> by a <code>null</code> value.
   *
   * @param signature the signature of the new theory.
   * @return a response code.
   */
  /*@ public normal_behavior
//    @   requires prover_started;
    @   ensures \result == ProverResponse.OK ||
    @           \result == ProverResponse.FAIL ||
    @           \result == ProverResponse.SYNTAX_ERROR;
    @*/

  public /*@ non_null @*/ ProverResponse signature(/*@ non_null @*/ Signature signature)

  /**
   * Declare a new axiom in the current theory.
   *
   * @param formula
   * @return a response code.
   */
  /*@ public normal_behavior
//    @   requires signature_defined;
    @   ensures \result == ProverResponse.OK ||
    @           \result == ProverResponse.FAIL ||
    @           \result == ProverResponse.SYNTAX_ERROR ||
    @           \result == ProverResponse.INCONSISTENCY_WARNING;
    @*/

  public /*@ non_null @*/ ProverResponse declare_axiom(/*@ non_null @*/ Formula formula)

  /**
   * Make an assumption.
   *
   * @param formula the assumption to make.
   * @return a response code.
   */
  /*@ public normal_behavior
//    @   requires signature_defined;
    @   ensures \result == ProverResponse.OK ||
    @           \result == ProverResponse.FAIL ||
    @           \result == ProverResponse.SYNTAX_ERROR ||
    @           \result == ProverResponse.INCONSISTENCY_WARNING;
    @*/

  public /*@ non_null @*/ ProverResponse make_assumption(/*@ non_null @*/ Formula formul
  /**
   * Retract some assumptions.
   *
   * @param count the number of assumptions to retract.
   * @return a response code.
   */
  /*@ public normal_behavior
    @   requires count >= 0 ;
    @   ensures \result == ProverResponse.OK ||
    @           \result == ProverResponse.FAIL;
    @*/

  public /*@ non_null @*/ ProverResponse retract_assumption(int count);

  /**
   * Check the validity of the given formula given the current theory,
   * its axioms, and the current set of assumptions.
   *
   * @param formula the formula to check.
   * @param properties a set of hints, primarily resource bounds on
   * this validity check.
   * @return a response code.
   *
   * @design kiniry 30 June 2004 - Possible alternative names for this
   * method include check(), is_entailed(), is_an_entailed_model_of().
   * I prefer is_valid().
   */
  /*@ public normal_behavior
    @   ensures \result == ProverResponse.YES ||
    @           \result == ProverResponse.NO ||
    @           \result == ProverResponse.COUNTER_EXAMPLE ||
    @           \result == ProverResponse.SYNTAX_ERROR ||
    @           \result == ProverResponse.PROGRESS_INFORMATION ||
    @           \result == ProverResponse.TIMEOUT ||
    @           \result == ProverResponse.NO ||
    @           \result == ProverResponse.FAIL;
    @*/

  public /*@ non_null @*/ ProverResponse is_valid(/*@ non_null @*/ Formula formula,▮
                                                  Properties properties);

  /**
   * Stop the prover.
   *
   * @return a response code.
   */
  /*@ public normal_behavior
    @   ensures \result == ProverResponse.OK ||
    @           \result == ProverResponse.FAIL;
    @*/

  public /*@ non_null @*/ ProverResponse stop_prover();
```

```
}
```

The response codes sent by the prover to the front-end are specified in the Java class `ProverResponse`.

```
import java.util.Properties;

class ProverResponse
{
    /**
     * A singleton response code to indicate everything is fine.
     */
    public /*@ non_null @*/ static ProverResponse OK =
new ProverResponse();

    /**
     * A singleton response code to indicate that something is seriously
     * wrong with the corresponding call and/or the prover and a failure
     * has taken place.  A response code of FAIL typically indicates a
     * non-correctable situation.  The {@link #info} field should be
     * consulted for additional information, and no further calls, but
     * for {@link ProverInterface#stop_prover()} should be made.
     */
    public /*@ non_null @*/ static ProverResponse FAIL =
new ProverResponse();

    /**
     * A singleton response code to indicate a positive response to the
     * last command.
     */
    public /*@ non_null @*/ static ProverResponse YES =
new ProverResponse();

    /**
     * A singleton response code to indicate a negative response to the
     * last command.
     */
    public /*@ non_null @*/ static ProverResponse NO =
new ProverResponse();

    /**
     * A singleton response code to indicate a counter-example is
     * available.  The counter-example is contained in the {@link
     * #formula} field of this response object.
     */
    public /*@ non_null @*/ static ProverResponse COUNTER_EXAMPLE =
new ProverResponse();

    /**
     * A singleton response code to indicate a syntax error in the
     * corresponding prover call.  The {@link #info} field should be
     * consulted for additional information.
     */
    public /*@ non_null @*/ static ProverResponse SYNTAX_ERROR =
new ProverResponse();

    /**
     * A singleton response code to indicate that some progress
     * information is available from the prover.  The {@link #info}
     * field should be consulted for additional information.
     */
    public /*@ non_null @*/ static ProverResponse PROGRESS_INFORMATION =
new ProverResponse();

    /**
     * A singleton response code to indicate that the prover has timed
     * out on the corresponding prover call.  The {@link #info} and/or
     * {@link #formula} fields should be consulted for additional
     * information.
     */
    public /*@ non_null @*/ static ProverResponse TIMEOUT =
new ProverResponse();

    /**
     * A singleton response code to indicate an inconsistency warning
     * from the prover for one or more of the previous {@link
     * ProverInterface#declare_axiom(Formula)} and {@link
     * ProverInterface#make_assumption(Formula)} calls.  The {@link
     * #info} and/or {@link #formula} fields should be consulted for
     * additional information.
     */
    public /*@ non_null @*/ static ProverResponse INCONSISTENCY_WARNING =
new ProverResponse();

    /**
     * A formula.  Typically, this field is used to represent a
     * counter-example in response to a call to {@link
     * ProverInterface#is_valid(Formula)}.
     */
    public Formula formula;

    /**
     * A set of properties.  Typically, this field is used to represent
     * property/value pairs specific to reporting prover progress,
     * state, resource utilization, etc.
     */
    public Properties info;

    /**
     * A private constructor that is only to be used during static
     * initialization.
     */
    protected ProverResponse() {
; // skip
    }
```

```
        // placeholder for factory for building ProverResponses
        /*@   ensures \result == ProverResponse.OK ||
          @   \result == ProverResponse.FAIL ||
          @   \result == ProverResponse.YES ||
          @   \result == ProverResponse.NO ||
          @   \result == ProverResponse.COUNTER_EXAMPLE ||
          @   \result == ProverResponse.SYNTAX_ERROR ||
          @   \result == ProverResponse.PROGRESS_INFORMATION ||
          @   \result == ProverResponse.TIMEOUT ||
          @   \result == ProverResponse.INCONSISTENCY_WARNING;
          @*/
        static public ProverResponse factory(int return_code) {

    /*
     * Naive implementation, should be redefined in subclasses
     */

    if( return_code >= 0 )
        return ProverResponse.OK;
    else
        return ProverResponse.FAIL;

        }
    }
```

The classes `Formula` and `Signature` are still to be defined.

Index

A

API 3

B

Built-in Theories 6

C

Copying 8

E

ESC/Java2 Prover Built-in Theories 6

I

Introduction 2

P

Prover API 3
Prover Built-in Theories 6

R

References 7

T

The ESC/Java2 Prover API 3
Theoretical Assumptions 6
Theories 6