# Making Maude Definitions more Interactive

Andrei Arusoaie[1], Traian Florin Șerbănuță[1,2],
Chucky Ellison[2], and Grigore Roșu[2]

[1] University Alexandru Ioan Cuza of Iași
{andrei.arusoaie,traian.serbanuta}@info.uaic.ro
[2] University of Illinois at Urbana-Champaign
{tserban2,celliso2,grosu}@illinois.edu

**Abstract.** This paper presents an interface for achieving interactive executions of Maude terms by allowing console and file input/output (I/O) operations. This interface consists of a Maude API for I/O operations, a Java-based server offering I/O capabilities, and a communication protocol between the two implemented using the external objects concept and Maude's TCP sockets. This interface was evaluated as part of the $\mathbb{K}$ framework, providing interactive interpreter capabilities for executing and testing programs for multiple language definitions.

## 1 Introduction

Formal specifications are often used when designing complex systems. The executability aspect of rewriting logic [11], and Maude's ability to efficiently execute, explore, and analyze rewrite theories [5] offers an additional level of support when designing a new system, as it allows the designer to experiment, test, and revise a specification before deciding to implement it. In certain cases, it even allows for the specification to become the implementation, eliminating the need of building another executable model. However, many systems include a human component, who is allowed/required to provide input to the system for directing its evolution.

To handle interaction, Maude provides the read-eval-print loop in the LOOP-MODE standard module, but this allows only for very limited user interaction. Furthermore, according to the Maude manual [4], it "may not be maintained in future versions, because the support for communication with external objects makes it possible to develop more general and flexible solutions for dealing with input/output in future releases." While Maude's external objects do allow interaction in principle, they are low-level and cannot be used for generic input/output (I/O) without significant infrastructure external to Maude. Our contribution is to provide this infrastructure and to develop an easy to use interface for it within Maude.

Figure 1 presents a high-level view of the I/O interface. It includes a Maude API for I/O, a Java-based server for handling requests, and a protocol for delivering queries and transmitting the responses. Using this interface, potentially any Maude definition can be enhanced with I/O capabilities. A Java wrapper which runs on top of both Maude and the Java server allows for the user to

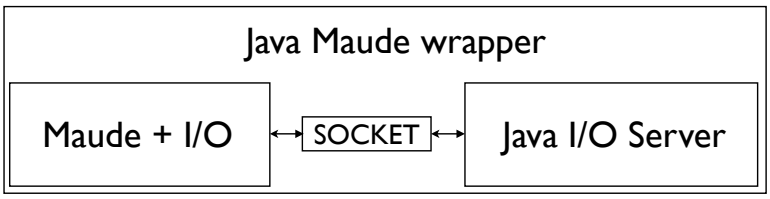


**Fig. 1.** The architecture of the Maude I/O interface

experience interpreter-like behavior: the console in which the program is started is the one interactively displaying the output of the program and requesting input. In addition to console I/O, this new framework also provides support for file I/O, including both sequential and random-access. Moreover, all of these actions are allowed to take place potentially anywhere in the term.

The Maude I/O interface and the examples presented in this paper, as well as the Java I/O server are available online [3,2]. Additionally, this interface has been used as part of the $\mathbb{K}$ framework tool [1] to obtain interpreter-like behavior for a number of language definitions, including a definition of the C language [8].

The remainder of this paper is structured as follows. Section 2 describes the I/O interface from a user point-of-view and illustrates some usage patterns through examples. Section 3 details the implementation, both for the Maude I/O client and the Java I/O server components. Section 4 reviews related work and Section 5 concludes.

## 2 The I/O interface

The basic standard I/O interface exposes several I/O commands for the standard input and standard output streams defined in the STANDARD-IO module:

```
op #printString : String → IOResult .
op #readInt() : → IOResult .
op #eof() : → IOResult .
op #printChar : Char → IOResult .
op #readChar() : → IOResult .
op #readToken() : → IOResult .
```

The resulting sort for these directives is "IOResult" which is defined in the "IO-INTERFACE" together with several constructors for it:

```
op #success : → IOResult .
op #string : [String] → IOResult .
op #int : Int → IOResult .
op #char : [Char] → IOResult .
op #ioError : String → IOResult .
op #flag : [Bool] → IOResult .
op #eof : → IOResult .
```

`#printString` sends an entire string, character by character, to the standard output stream and returns `#success`. `#readInt` reads a token and returns an `#int` containing the number read. `#eof()` tests the standard input stream for the end of file and returns a `#flag` result with the argument set appropriately. `#printChar` sends one character to the standard output stream and returns `#success`. `#readChar` reads a character from the standard input stream and returns

a `#char` or `#eof`. Finally, `#readToken` skips the whitespace from the standard input and then returns a `#string` containing all characters read until the next whitespace or the end of file is encountered, or `#eof` if the end of file is reached while skipping over whitespace. In case of an I/O error or a communication error, an `#ioError` term detailing the error is produced. We will present their formal definition in the next section, but in the meanwhile, let us start with some examples.

### 2.1 Example 1: A straightforward usage of the I/O interface

Let us begin with an example showing how our I/O interface can be used in a Maude definition. For this we have chosen a very simple expression language, called EXP. The following module defines its syntax:

```
mod EXP−SYNTAX is
  including INT .
  including STRING .
  sort Exp .
  subsort Int < Exp .
  op _+_ : Exp Exp →Exp [ditto] .
  op _*_ : Exp Exp →Exp [ditto] .
  op _ifnz_ : Exp Exp → Exp [strat(2 0)] .
  op nzloop : Exp → Exp [strat (0)] .
  op input : String → Exp .
  op print : String Exp → Exp .
endm
```

EXP has integers as basic values and extends two of the integer operations: addition and multiplication. Moreover, it provides a guarded expression, `ifnz`, which evaluates its first argument only if the evaluation of the second one produces a non-zero value, and a fix-point operator, `nzloop`, which evaluates its argument as long as its value is not zero. Note that in the absence of any side effects, the `nzloop` construct is rather non-sensical, as its argument would always evaluate to the same value. However, adding I/O constructs to the language allows for some interesting (albeit simple) programs to be written in this language, like, for example:

nzloop(print("3∗x=",3 ∗ input("x= (0 to stop)? ")))

The intended meaning of `input` construct is somehow similar to the `INPUT` instruction of the `BASIC` language [6], in that it prompts the string in its argument to the user and expects an integer to be entered, returning that integer. The meaning of `print` is that it prints the string in the first argument, then prints the string representation of the second argument (which is expected to be evaluated to an integer), and then advances the line feed. With this intuition in mind, the semantics of the program above is that reads a number "x" from the console and computes and displays "3*x" until the number entered is 0 (included).

The module in Figure 2 formally defines the semantics described above. Assuming this module is included in a file named `io-test.maude` (which also loads the `io-interface.maude` file) and that the Maude command for rewriting (with external objects) the above program is written in a file named `io-test-cmd.maude`, the following command "executes" the program interactively:

```
java -jar MaudeIO.jar \
          --maudeFile io-test.maude \
```

```
mod EXP-SEMANTICS is
  including EXP-SYNTAX . including STANDARD-IO .
  op read : -> Exp .

  eq input(S)
   = #printString(S);
     #readInt();
     read .

  eq nzloop(E) = nzloop(E) ifnz E .
  eq E ifnz 0 = 0 .
  eq E ifnz NzI = E .

  eq print(S,I)
   = #printString(S + string(I,10) + "\n");
     I .

  op _;_ : IOResult Exp -> Exp
                                  [strat (1 0)] .
  eq #success ; E = E .
  eq #int(I) ; read = I .

  var I : Int . var S : String . var NzI : NzInt . var E : Exp .
endm
```

**Fig. 2.** A Straight-forward semantics for the EXP language

```
--moduleName EXP-SEMANTICS \
--commandFile io-test-cmd.maude
```

Here is a possible interaction sequence between the user and the tool:

```
x= (0 to stop)? 5
3*x=15
x= (0 to stop)? 10
3*x=30
x= (0 to stop)? 7
3*x=21
x= (0 to stop)? 0
3*x=0
Maude> ========================================
erewrite in KRUNNER : nzloop(print("3*x=", 3 * input("x= (0 to stop)? "))) .
rewrites: 8452 in 59ms cpu (5345ms real) (141321 rewrites/second)
result Zero: 0
Maude> Bye.
```

Allowing I/O operations anywhere in the term/program provides a high degree of flexibility, but at a price. As running multiple I/O commands at the same time creates race conditions, the user has to provide mechanisms to avoid these races. One such example is the "`_;_`" command defined in the semantics above whose only purpose is to ensure that the printing command completes before the next command is executed (enforced by the strategy annotation), resembling the `_>>_` sequentialization operator of the Haskell I/O monad [13].

However, parallel I/O commands are still possible in our language, producing potentially undesirable effects. For example, when executing the program

```
print("Hello ",1) + print("World!",2)
```

a possible result would be the following:

```
HWoerllldo! 2
1
Maude> ========================================
erewrite in KRUNNER : print("Hello ", 1) + print("World!", 2) .
rewrites: 1525 in 9ms cpu (23ms real) (160374 rewrites/second)
result NzNat: 3
```

While this may be acceptable behavior, it is also possible to avoid it if not. This can be done by sequentializing the two printing expressions programatically, e.g., by relying on the strategy of `ifnz`, like "`print("World!",2) ifnz print("Hello ",1)`".

### 2.2 Interaction with Maude's analysis tools

The definition presented above is very simple, but it can also easily become quite chaotic, due to its very direct use of I/O (through external objects). It is also impossible to test it in the absence of the I/O server. However, it is quite easy to add I/O as a natural extension to specifications which are already amenable for testing, exploration, and analysis.

Figure 3 presents an SOS-style rewriting logic semantics [17,12] for the EXP language. To simulate input and output it uses a configuration which, in addition to the program to be evaluated, contains a list of integers for input and a string buffer for output. The small-step semantics is given through the rules for the $\bullet\_$ operation which defines the application of a single computation step in an SOS style. Also note that the semantics of `input` has changed, by creating separate small steps for the prompt and read operations.

Given an input list, this semantics can be tested, explored, and even model-checked using Maude. Moreover, using the proposed I/O interface, it is quite easy to turn it into an I/O-enabled interpreter. One relatively simple way to achieve this is by adding a special input constant `requestInt` and an equation for requesting an int when the input list becomes empty:

**op** requestInt : $\rightarrow$ [Int] .
**rl** $\bullet${read, nil , Out} $\Rightarrow$ {read,requestInt , Out} .

The reason for allowing `requestInt` to be in the kind of `List{Int}` is because we want to catch this state at the top level by making the equation for "$*$" not applicable. However, to ensure that `requestInt` is propagated, we need to declare the variable `In'` to also be in the kind. Then, we need to add an additional equation at the top to flush the output buffer and perform the actual read:

**eq** $*${E,requestInt,Out} = $* \bullet${(#printString(Out) ; #readInt() ; {E,nil,""})} .

These operations are sequenced using the same idea as in the previous definition; moreover, once an integer is read, it is added to the `In` list:

**op** _;_ : IOResult State $\rightarrow$ State [**strat** (1 0)] .
**eq** #success ; State = State .
**eq** #int(I) ; {E,In,Out} = {E,I In,Out} .

```
mod EXP−SEMANTICS is
  including EXP−SYNTAX . including LIST{Int} . including STANDARD−IO .

  sort State .                    op {_,_,_} : Exp List{Int} String → State .

  op •_: State ⇝ State .          op ∗_ : State ⇝ State .
  eq ∗ State = ∗ • State .              op read : → Exp .

crl •{E1 ∗ E2,In,Out} ⇒ {E1' ∗ E2,In',Out'}
 if •{E1,In,Out} ⇒ {E1',In',Out'} .
crl •{E1 + E2,In,Out} ⇒{E1' + E2,In',Out'}
 if •{E1,In,Out} ⇒ {E1',In',Out'} .
crl •{print(S,E),In,Out} ⇒ {print(S,E'),In',Out'}
 if •{E,In,Out} ⇒ {E',In',Out'} .
 rl •{input(S),In,Out} ⇒ {read,In,Out + S} .
 rl •{read,I In, Out} ⇒ {I, In, Out} .
 rl •{print(S,I),In,Out} ⇒ {I,In,Out + S + string(I,10) + "\n"} .
crl •{E2 ifnz E1,In,Out} ⇒ {E2 ifnz E1',In',Out'}
 if •{E1,In,Out} ⇒ {E1',In',Out'} .
 rl •{E ifnz 0,In,Out} ⇒ {0,In,Out} .
 rl •{E ifnz NzI,In,Out} ⇒ {E,In,Out} .
 rl •{nzloop(E),In,Out} ⇒ {nzloop(E) ifnz E,In,Out} .

  var I : Int . var S : String . var NzI : NzInt . var E E1 E2 E' E1' E2' : Exp .
  var In In' : List{Int} . var Out Out' : String . var State : State .
endm
```

**Fig. 3.** An SOS-like semantics for EXP in Maude

Finally, we might want to add an additional equation to flush the output buffer at the end of the execution:

```
eq ∗ •{I,In,Out} = #printString(Out) ; {I,In,""} .
```

Since in this particular example the communication with the I/O server is enforced to occur at the top of the term, this semantics guarantees the sequencing of the print statements above, producing a reasonable output:

```
Hello 1
World!2
Maude> ========================================
erewrite in KRUNNER : * {print("Hello ", 1) + print("World!", 2),nil,""} .
rewrites: 1455 in 12ms cpu (51ms real) (114738 rewrites/second)
result State: {3,nil,""}
```

Moreover, if we exclude the final equation from this semantics (e.g., by putting it in its own separate module which is only used for I/O interpretation) the modified semantics still has all the good properties of the non-I/O definition. That is, if provided with enough input in the initial configuration, the list will

not become empty during the rewrite/search/model-checking process and thus the I/O-blocking communication commands will not impede the analysis process.

*Note:* As the I/O operations rely on external objects, they cannot be executed as part of the rewriting process for verifying a rewriting condition. Therefore in definitions using conditional rewriting, like the one presented above, the I/O request must be pushed outside of the condition context before it can be handled (we did that above by using the `requestInt` construct).

### 2.3 The File I/O Interface

As detailed in the next section, the simple I/O interface exhibited above is implemented in terms of a file I/O interface, defined by the IO-INTERFACE module, which provides file-handle-parameterized versions of its basic I/O commands:

```
op #fPrintString : Nat String → IOResult .   op #fPrintChar : Nat Char → IOResult .
op #fReadInt : Nat → IOResult .              op #fReadChar : Nat →IOResult .
op #fEof : Nat → IOResult .                  op #fReadToken : Nat →IOResult .
```

In addition to those, it also provides commands for opening and closing files,

```
op #open : String → IOResult .          — opens a file , mapping it to a #handle
op #reopen : Nat String → IOResult .    — maps a different file to the #handle
op #close : Nat → IOResult .            — closes the file mapped to the #handle
```

as well as several lower-level commands for accessing the contents of a file:

```
op #fReadByte : Nat →IOResult .      op #flush : Nat → IOResult .
op #fPutByte : Nat Nat →IOResult .   op #tell : Nat → IOResult .
op #fPeekByte : Nat →IOResult .      op #seek : Nat Nat → IOResult .
```

As a simple example of how this more advanced interface could be used, let us provide an addition to the I/O rules above, which would allow specifying the input file in the initial configuration. Thus, if an input file is specified, it would be used instead of the standard input stream whenever the input list becomes empty.

To achieve this we introduce a new construct, `stream`, a rewriting rule, and an equation. `stream` holds a file handle and acts as a potential marker in the input list to signal where input should be read from. The first rule signals a request for reading from the file stream, while the second one actually performs the read operation.

```
op stream : IOResult → List{Int} .

rl •{read,stream(#handle(N)),Out}
⇒ {read,requestInt stream(#handle(N)),Out + S} .

eq ∗{E,requestInt stream(#handle(N)),Out}
 = ∗ .(#printString(Out) ; #fReadInt(N) ; {E,stream(#handle(N)),""}) .
```

With only this addition, the program behaves as follows. As long as there is any integer in the input list, the execution proceeds as without any I/O. When the

list contains no more integers, there are two options: (1) if the list is empty, then input will be requested from the standard input stream; (2) if the list contains a `stream` term, then input will be read from that stream.

For example, assuming the contents of the `test.in` file are "3 -5 0", rewriting with the following command:

```
erew * {nzloop(print("3*x=",3 * input("x= (0 to stop)? "))),
       stream(#open("file:test.in#r")),
       ""} .
```

will produce the output:

```
x= (0 to stop)? 3*x=9
x= (0 to stop)? 3*x=-15
x= (0 to stop)? 3*x=0
Maude> ========================================
erewrite in KRUNNER : * {nzloop(print("3*x=", 3 * input("x= (0 to stop)? "))),
stream(#open("file:test.in#r")),""} .
rewrites: 6311 in 39ms cpu (63ms real) (159962 rewrites/second)
result State: {0,stream(#handle(3)),""}
Maude> Bye.
```

The argument of `#open` contains the usual URI description of a file location before the "#" symbol, while the "r" after the symbol specifies that the file should be opened for read-only access.

### 2.4 Separating the I/O server from Maude

The common usage pattern for the I/O interface presented above is that both Maude and the Java I/O Server are executed as subprocesses of the same Java wrapper application. This way, users interact directly with the console they used to start the execution of the program. Moreover, as this close integration uses a fresh TCP port for communication, this allows multiple instances of Maude using I/O to be running at the same time. Unfortunately this hides the Maude console and inhibits the user from interacting with Maude directly. As sometimes it might be useful to have both the I/O console and the Maude console available, let us describe how this can be achieved.

First, one needs to fix the port on which communication takes place. To do so, it needs to add an equation of the form "`eq #TCPPORT = 8001 .`" either in the definition using the I/O interface, or right after the definition of the `#TCPPORT` constant in the `tcp-interface.maude` file (if this behavior is desired for all definitions). Then, the server must be started first in a console taking as parameter the same port number:

```
java -jar ioserver.jar 8001
```

Once the server is running, the definition can be loaded into Maude in a separate console. Now we have two consoles, communicating between them. The rewriting commands can be given using the normal Maude console and the Maude result will be displayed here, while the I/O messages will be displayed in the console running the I/O server and I/O input will be requested from there.

## 3 Implementation

This section comes to give additional details for the implementation of the I/O interface described in the prior sections.

As hinted in the introduction and depicted in Figure 1, both the I/O Server and Maude are wrapped by a Java process which is intended to offer to the user the console he expects when a program is executed. The wrapper hides some operations that a casual user would not care about. This wrapper is executed as a normal Java jar, taking as arguments the file containing the Maude definition, the name of the main module, and the file containing the rewrite command to be executed, as exhibited in Section 2.1. The main purpose of this wrapper is to automatically setup Maude and the I/O Server. An unused TCP port $p$ is identified and an equation:

**eq** #TCPPORT = $p$ .

is added automatically in a module KRUNNER, which includes the main module of the definition. The wrapper also sets up the port number for the I/O Server which is then started before Maude is launched and the files containing the modified definition and the rewriting command are loaded.

### 3.1 The Maude I/O client

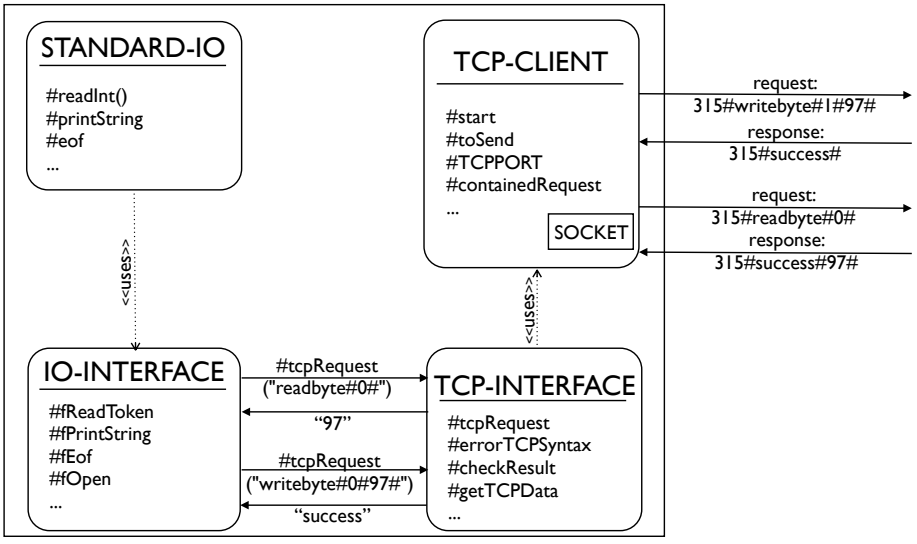


**Fig. 4.** The architecture of the Maude component of the I/O interface

Figure 4 presents the architecture of the Maude component of the I/O interface and briefly describes the interaction between its various sub-components. The

user interacts with it using either the basic, console-only interface provided by the STANDARD-IO module (see Section 2.1 and 2.2), or the more comprehensive, file-based one provided by the IO-INTERFACE module (see Section 2.3). The constructs in the STANDARD-IO module desugar into their correspondents from the IO-INTERFACE module. The IO-INTERFACE module reduces all operations to simple, byte-based requests, and uses the TCP-INTERFACE function `#tcpRequest` as an interface to the I/O Server. This result of communication is further interpreted by the functions in the IO-INTERFACE module and is transformed into a term of the IOResult sort which contains a series of constructors for each specific type of answers, such as `#success`, `#int`, `#string`, and `#eof`.

As most of the I/O interface was exhibited in the previous section, we will focus here on the implementation details: how the high-level constructs are expressed in terms of the lower level ones and how the communication takes place.

*The STANDARD-IO module* defines handles for the standard input, output, and error streams:

```
ops #stdin #stdout #stderr : → Nat .
eq #stdin = 0 .    eq #stdout = 1 .    eq #stderr = 2 .
```

These handles are then used to express the STANDARD-IO constructs in terms of those from the IO-INTERFACE:

```
eq #eof() = (#fEof(#stdin)) .
eq #readChar() = (#fReadChar(#stdin)) .
eq #printChar(C) = #fPrintChar(#stdout,C) .

eq #readToken() = (#fReadToken(#stdin)) .
eq #readInt() = #fReadInt(#stdin) .
eq #printString(S) = #fPrintString(#stdout,S) .
```

*The IO-INTERFACE module* provides functionality for lower level I/O constructs falling the following three categories. The first category consists of I/O primitives whose semantics is simply a request to the server:

```
eq #open(S) = #handle(rat(#tcpRequest("open#" + S + "#"), 10)) .
eq #close(N) = #checkSuccess(#tcpRequest("close#" + string(N,10) + "#")) .
eq #fReadByte(N)
 = #byte(rat(#tcpRequest("readbyte#" + string(N,10) + "#"),10)) .
ceq #fPutByte(N,B)
  = #checkSuccess(#tcpRequest(
                    "writebyte#" + string(N,10) + "#" + string(B,10) + "#"))
 if B < 256 .
```

These functions rely on functions like `#checkSuccess` to translate the string answer provided by `#tcpRequest` into a term of the appropriate IOResult type. Note that these TCP requests have a very regular form, of #-separated pieces of information, among which the first is the command, and the following are additional arguments to the command, like, e.g., "`open#file:in.txt#r#`".

The second category includes functions that easily desugar into primitives, e.g.:

```
eq #fPrintChar(N,C) = #fPutByte(N,ascii(C)) .
eq #fReadChar(N) = #char(#fReadByte(N)) .
eq #fReadInt(N) = #int(#fReadToken(N)) .
```

```
op #char : IOResult → IOResult .
eq #char(#byte(B)) = #char(char(B)) .
eq #char(#eof) = #eof .
```

```
op #int : IOResult → IOResult .
eq #int(#string(S)) = #int(rat(S,10)) .
eq #int(#eof) = #eof .
```

Finally, there are the more involved functions like `#fPrintString` which iterates over the characters of a string one by one, waiting for the printing of a character to succeed before printing the next and flushing the output buffer at the end,

```
eq #fPrintString(N,"") = #flush(N) .
eq #fPrintString(N,S)
 = #fPrintString(N,S, #fPrintChar(N,substr(S,0,1))) [owise] .
op #fPrintString : Nat String IOResult → IOResult .
eq #fPrintString(N,S,#success) = #fPrintString(N,substr(S,1,length(S))) .
```

or like `#fReadToken` which reads character by character from the file specified by the handle, skipping over the initial whitespace and then accumulating the characters read until whitespace is again encountered.

*The TCP-INTERFACE module* provides functionality for initializing the communication process and extracting the relevant data once the communication process concludes with a result. The semantics of the `#tcpRequest` construct is:

```
eq #tcpRequest(S:String) = #tcpRequest(S:String, counter) .
```

```
op #tcpRequest : String [Nat] → String .
eq #tcpRequest(S:String, N:Nat)
 = #checkResult(#containedRequest(#start(N:Nat) #toSend(S:String))) .
```

`#tcpRequest` creates a object configuration, wrapped by the `#containedRequest` construct; this configuration includes a primitive to start the communication with the server and the request to be sent. `#checkResult` expects the result to be either of the form `success#data###` or of the form `fail#reason###` and returns "`data`" in case of success or an `#errorTCPSyntax` term otherwise.

*The TCP-CLIENT module* provides rules for initiating the TCP communication through a socket, sending a message, waiting for a response, and closing the connection. First, a fresh id is generated using Maude's builtin COUNTER module, and is used to establish a connection with the I/O server using the TCP sockets interface provided by Maude:

```
op #start : → Configuration .
op #start : Nat → Configuration .
```

```
eq #start = #start(counter) .
op cnum : Nat → Oid .
```

```
rl #start(N)
⇒ <>      < cnum(N) : Client | state: connecting >
   createClientTcpSocket(socketManager, cnum(N), "localhost", #TCPPORT) .
```

The `#TCPPORT` constant is the one mentioned in Section 2.4, being either set manually if running the I/O server separately from Maude, or automatically by the Java wrapper.

Once the initial exchange of messages takes place between Maude and the server, the `#toSend` message is transformed into a message addressed to the server and the state of the system becomes `sending`:

```
rl < cnum(N) : Client | state: connected,  connectedTo: Server,  A > #toSend(S)
⇒ < cnum(N) : Client | state: sending,  connectedTo: Server,  A >
    send(Server, cnum(N), (string(N:Nat,10) + "#" + S + "\r\n")) .
```

Note that the body of the message is prefixed with the number of the client (for logging and debugging reasons) and is appended with the end-of-line markers as a message separator. Thus, the complete message being sent to the server is of the form "`23#open#file:in.txt#r#\r\n`".

There are additional rules for continuing the dialogue with the server following the TCP protocol, but once the communication has finished, the answer is extracted by the rule below and the header of the message (containing the number identifying the communication) is removed by the `#checkAnswer` function:

```
rl #containedRequest(<> < Me : Client | state: finished,  answer: S,  A >)
⇒ #checkAnswer(Me, S) .
```

### 3.2 The Java I/O server

In this section we discuss the I/O Server architecture and implementation details. While describing the main components, we will motivate our choices regarding their design. The purpose of the I/O Server is to implement a socket-based service for simulating file operations over regular files, standard input, and standard output. So far it has only been used with Maude as a client, but the implementation is rather client-independent.

The architecture and data flow of the I/O Server is depicted in Figure 5. The server has two main components: the communication component, which is responsible for receiving/sending messages, and the resource management component, which manages resources (files, stdin, stdout, stderr) and operations on them. In this section, we will refer to them as RequestManager and ResourceManager.

Before describing these components, we will briefly explain the behavior and capabilities of the I/O Server. First, the complete list of operations currently accepted by the server is the following:

- *open* - open a new file
- *close* - close a file or stream
- *readbyte* - read a byte
- *writebyte* - write a byte
- *flush* - flush the buffer
- *reopen* - reopen an existent file
- *seek* - seek a particular location
- *position* - go to a specific location
- *peek* - peek a byte
- *eof* - check end of file

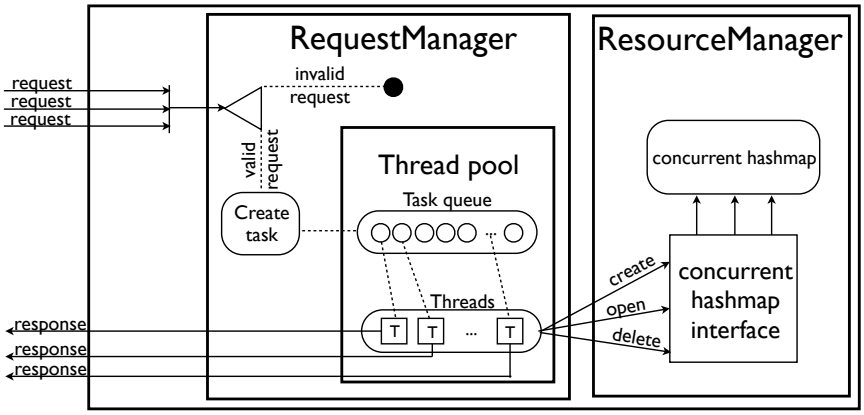


**Fig. 5.** The data-flow scheme of the I/O server.

The server receives requests, each request carrying one of the commands above, and it answers the requests upon executing the corresponding operation. The requests and answers are formatted as a string which contains data separated by "#":

request: 315#writebyte#1#97#
response: 315#success#

The request contains the client id, the operation name followed by the resource id and operation parameters, if any. For instance in the above request the client having the id 315 asks the server to write at the standard out stream the byte corresponding to character "a" (the stdin, stdout and stderr streams have fixed ids 0, 1, and 2, respectively). The I/O Server generates a fresh id (identifying the file handle) as response to the client which requests to "open" a new file. The response usually contains the client id which stands for a weak kind of authentication, the status of the operation and the result of its execution if exists.

The RequestManager component uses TCP sockets to provide a reliable point-to-point communication with the client. To be able to handle multiple request concurrently, we use the *Thread Pool* pattern. For each request, the associated command is analyzed and, if found valid, a task is created and queued for execution. Commands are executed in parallel, each thread being responsible for executing a command and sending the response to the client. The thread pool executor (defined in the `ThreadPoolExecutor` class) registers commands as members of the abstract class `Command` which implements the Java `Runnable` interface, and assigns them to threads as these become available.

The second component of the I/O Server handles resources and operations on them. Currently, the ResourceManager can store three types of resources: random access files, standard input, and standard output. It provides operations to add, retrieve, or delete a resource. Some operations, for instance *peek*, *readbyte*, *seek*, and *position* cannot be applied on the standard output; in such

cases, when operations are not applicable on a specific type of resource, the ResourceManager throws exceptions to the RequestManager, which in turn will send to the client a meaningful failure message.

Regarding the implementation, for each resource we have a corresponding class (`ResourceInFile`, `ResourceOutFile`, ...) which must extend the abstract class `Resource`. This class contains abstract methods which correspond to commands received by the I/O Server (`readbyte()`, `writebyte()`, ...). To store the resources we use the `ConcurrentHashMap` class which has two highly concurrent properties: writing to it locks only a portion of the map and reads can generally occur without locking.

## 4 Related Work

This paper uses a technology similar to that used for developing Mobile Maude [7], but with a different aim: there sockets communication was used to communicate between different instances of Maude; here we use a similar mechanism to communicate with an external I/O server.

The motivation for this work came from our research in programming language design, namely in our efforts to make the implementation of the $\mathbb{K}$ semantic framework [1,16] easier to use and experiment with. $\mathbb{K}$ [16] is a rewrite-based executable semantic framework specialized for defining programming languages, type systems and formal analysis tools. The $\mathbb{K}$ tool [1,18] transforms $\mathbb{K}$ definitions into rewriting logic theories which can be executed, explored and analyzed using Maude. So far the $\mathbb{K}$ tool has been used to give complete definitions to real languages like C [8] and Scheme [10], along with many educational languages and a novel rewriting-based program verification logic named matching logic [15,14]. The I/O interface described in this paper is an integral part of the $\mathbb{K}$ tool and provides I/O capabilities for all $\mathbb{K}$ semantics defined using the tool. Most notably, it has been used to extensively test the $\mathbb{K}$ definition of C [8] against the gcc torture tests [9].

## 5 Conclusions and Future Work

We have described a methodology and a system for achieving interactive (file) input/output from within the Maude system. This technology was designed for modeling, in an executable way, runtime interaction needed in a system. It can additionally be used for runtime logging or tracing, and provides an easy way of getting output from a Maude execution. The I/O interface is generic and can easily be used in potentially any Maude definition. The interface itself is reasonably stable, as it has been implemented and extensively used as part of the $\mathbb{K}$ framework.

This work could serve as a means for experimenting with I/O before the technology is integrated in Maude directly as a special purpose external object. As our interface uses URIs, it should be relatively easy to incorporate support for accessing additional resources such as URLs. Moreover, adding primitives for locking resources would offer a thread-safe mechanism of accessing the resources.

## 6 Acknowledgments

The research presented in this paper was supported in part by the DAK project Contract 161/15.06.2010, SMIS-CSNR 602-12516.

## References

1. The K semantic framework website (2010), https://k-framework.googlecode.com
2. The Java I/O server: svn repository with full sources (2011), https://k-framework.googlecode.com/svn/trunk/core/java
3. The Maude I/O interface: svn repository with specification and examples (2011), http://k-framework.googlecode.com/svn/branches/inProgress/core/io
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.6) (January 2011), http://maude.cs.uiuc.edu/maude2-manual/
5. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude, A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007)
6. Coan, J.S.: Basic BASIC: an introduction to computer programming in BASIC language. Hayden Book Co., Rochelle Park, N.J. (1978)
7. Durán, F., Riesco, A., Verdejo, A.: A distributed implementation of Mobile Maude. In: WRLA'06. ENTCS, vol. 176 (4), pp. 113–131 (2007)
8. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL'12. ACM (2012), to appear
9. FSF: C language testsuites: "C-torture" version 4.4.2 (2010), http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html
10. Meredith, P., Hills, M., Roşu, G.: An executable rewriting logic semantics of K-Scheme. In: SCHEME'07. pp. 91–103 (2007)
11. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
12. Meseguer, J., Braga, C.: Modular rewriting semantics of programming languages. In: AMAST'04. LNCS, vol. 3116, pp. 364–378. Springer (2004)
13. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: POPL '93. pp. 71–84 (1993)
14. Rosu, G., Stefănescu, A.: Matching logic: A new program verification approach. In: ICSE'11 (NIER Track). pp. 868–871 (2011)
15. Roşu, G., Ellison, C., Schulte, W.: Matching logic: An alternative to Hoare/Floyd logic. In: AMAST'10. LNCS, vol. 6486, pp. 142–162 (2010)
16. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic and Algebraic Programming 79(6), 397–434 (2010)
17. Serbănuţă, T.F., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Information and Computation 207, 305–340 (2009)
18. Şerbănuţă, T.F., Roşu, G.: K-Maude: A rewriting based tool for semantics of programming languages. In: WRLA'09. LNCS, vol. 6381, pp. 104–122 (2010)