This report is a slightly revised version of the article appearing in the proceedings of *PANEL'92, XVIII Latin-American Conference of Informatics,* 31 August–5 September, 1992, Las Palmas de Gran Canaria, Spain. Citations should refer to the Proceedings (*not* to this report).

The Sun RPC Language Semantics

Martín Musicante*
Department of Computer Science, Aarhus University,
DK-8000 Aarhus, Denmark

September 1992

Abstract

A formal description of the Sun Remote Procedure Call Protocol is given.
The description is written using the Action Notation style of formal specification. Action Notation proves to be adequate to express the meaning of the Sun RP communication mechanism.

KEYWORDS: Formal Semantics, Action Notation, Remote Procedure Call.

Introduction

The Sun Remote Procedure Call Protocol [11], is regarded as a basis for the actual implementation of Remote Procedure Call (RPC) in many distributed computing systems, mainly in those of workstation networks, running UNIX[1].

*On leave from Universidade Federal de Pernambuco, Departamento de Informática, Caixa Postal 7851, CEP 50739 - Recife - PE, Brazil. This work was partly supported by CNPq, Brazil, grant No. 201.201/7-91, and DART, Denmark, grant No. 5.21.08.03. Author's E-mail address: `mam@daimi.aau.dk` or `mam@di.ufpe.br`.

[1]UNIX is a trademark of AT & T Bell Laboratories

The RPC concept [1] is a generalization of the procedure call concept, present in most imperative languages. The main idea is that the called procedure is executed in a different environment than the calling program, and possibly in a different machine. Data exchange between the caller and the called processes is performed only by parameter and return value passing. For a comprehensive explanation of the RPC concept, the reader is encouraged to read [1], [6] or [9].

The Sun RPC implementation scheme is based in the existence of a *port-mapper* process in each host. The port-mapper is accessible at a predefined, well-know address (the same for all systems) and its purpose is to keep a map of all the "remote accessible programs" available at the local machine.

The port-mapper itself is a remote program, consisting of several remote procedures. Operations such as registration or de-registration of programs, queries for the address of any available procedure in the machine, etc, can be performed using the port-mapper procedures. We refer to a program registered at the port-mapper as an RPC *server*. Each of the remote procedures is called a *service* available at the program. In the same way, a program that calls a remote procedure is said to be a *client*.

The descriptions given in [10] and [11] consist on the definition of a data description language (called XDR, External Data Representation language) and a program definition language (called the RFC Language). The RPC language [11] is an extension of the XDR language.

Each program can contain a set of program versions, and each version can be formed by several procedures (the actual remote procedures).

In this work, Action Notation [3] is used to give a formal semantics of the XDR and RPC languages. Modularity characteristics of Action Notation are used to provide the semantics of the RPC language as an extension of the XDR language specification. This is done in a way in which the XDR language semantics can be abstracted from the complete specification, if needed.

Both data representation and program definition languages follow the syntactic specifications given in [10, 11] (with minor changes to correct little errors and to give a better presentation for the semantic functions).

Action Notation

Action Notation [3] is a formal language designed to provide modular and readable descriptions of programming languages. Unlike denotational semantics, where *functions* are used to state the meaning of programs, Action Semantics uses special constructions called *actions*. Like in denotational semantics, semantic functions in Action Notation are defined in a *compositional* way, i.e., the meaning of each phrase of the language depends only on the meaning of its components.

Actions can be performed. They process information in a stepwise way. The performance of an action can yield four possible status: successful completion of execution (*complete*), exceptional termination (*escape*), unsuccessful completion of execution (*fail*) or non-termination (*diverge*).

The description language presented in [3] includes *basic actions* and *action combinators*, that is, rules to obtain new actions from simpler ones. Primitive actions to declare, store and access data and to manage processes and communication are presented also. As it is pointed out in [12], the set of combinators was carefully, chosen, in order to facilitate the task of stating the meaning of most programming languages.

Action descriptions are modular. This feature allows the modification of parts of the semantic description without affecting the whole system, as well as the reusability of modules. Action Notation modularity also allows the description to be organized in sections, very much in the same way as chapters and sections in a book.

1 Abstract Syntax

The abstract syntax of both the XDR and RPC languages is described in the following sections. The original description of the languages define some constructions that can be used as abbreviations for common patterns of definition. All these abbreviations were preserved in this description. We choose to consider them in the abstract syntax as a way of maintaining compatibility with the original descriptions.

Action Notation syntactic descriptions can be regarded as defining context-free languages in the same way well-known BNF's does. Usual regular expressions are allowed, as well.

1.1 XDR

The syntax of the XDR language is presented in this section. The description follows (with minor changes) the grammar given in [10].

The symbol '□' in the right hand side of the equations corresponding to Constant and Identifier indicates that the non-terminal is left undefined in this part of the specification.

grammar:

(1) Declaration =
  ⟦Type-specifier Identifier⟧ |
  ⟦Type-specifier Identifier "[" Value "]" ⟧ |
  ⟦Type-specifier Identifier "⟨" Value$^{?}$ "⟩" ⟧ |
  ⟦"opaque" Identifier "[" Value "]"⟧ |
  ⟦"opaque" Identifier "⟨" Value$^{?}$ "⟩" ⟧ |
  ⟦"string" Identifier "⟨" Value$^{?}$ "⟩" ⟧ |
  [Type-specifier "∗" Identifier⟧ | "void"

(2) Value = Constant | Identifier

(3) Type-specifier =
  ⟦"unsigned"$^{?}$ "int"⟧ | ⟦"unsigned"$^{?}$ "hyper"⟧ |
  "float" | "double" | "bool" | "void" |
  ⟦"enum" "{" Enumerand^{+} "}" ⟧ |
  ⟦"struct" "{" Declaration^{+} "}" ⟧ |
  ⟦"union" Union-body⟧ | Identifier

(4) Enumerand = ⟦Identifier "=" Valuen⟧

(5) Union-body = ⟦"switch" "(" Declaration ")" "{"
  Union-case^{+}
  ("default" ":" Declaration)$^{?}$

"}"⟧

(6) Union-case = ⟦"case" Value ":" Declaration⟧

(7) Constant-def = ⟦"const" identifier "=" Constant⟧

(8) Type-def =
⟦"typedef" Declaration⟧ |
⟦"enum" Identifier" {" Enumerand^{+} "}"⟧ |
⟦"struct" Identifier "{" Declaration^{+} "}"⟧ |
⟦"union" Identifier Union-body⟧

(9) Definition = Type-def | Constant-def

(10) Constant = □

(11) identifier = □

(12) Specification = Definition*

1.2 RPC

The RPC language is presented as an extension of the XDR language. The syntactic description follows (with minor changes) the grammar given in [11].

The semantics of an RPC specification supposes the existence of a procedure body to each procedure declared within a program. It also supposes that this procedure is bound to a name formed by the name of the declared program (but upper-cased), followed by an underscore character, and the version number which the referred procedure belongs to.

The symbol '□′ in the right hand side of Definition indicates that the non-terminal symbol is being extended at this point.

grammar:

needs: XDR

(1) Program-def =
$[\![$"program" Identifier "{" Version-def$^+$ "}" "=" Constant$]\!]$

(2) Version-def =
$[\![$"version" Identifier "{" Procedure-def$^+$ "}" "=" Constant$]\!]$

(3) Procedure-def =
$[\![$"Type-specifier" Identifier "(" Type-specifier ")" "=" Constant$]\!]$

(4) Definition = $\Box$ | Program-def

(5) Specification = Definition*

2 Semantic Functions

Bernantic functions are defined to establish the meaning of each phrase of the language. They are stated in a compositional way, by recursion on the syntactic structure of the language.

For space reasons, we present only a tiny but representative part of the semantic descriptions here. The interested reader can refer to the whole work, in [5].

2.1 XDR

This section is devoted to state the XDR language semantics. The definition here is concerned with two main issues: binding syntactic tokens to types and constants, and define translation functions for each type.

2.1.1 Elaborating Specifications

The main purpose of the semantic functions of this section is to create "bindings", i.e: associations from language tokens to semantic entities. The basic

binding action is the primitive "bind _ to _", which binds a syntactic token, given as first argument, to a value or cell given as second argument.

The combinator "_before_" takes two actions. The bindings produced by the first one are merged with the bindings received for the whole action. The result of this merge is passed to the second action (the one after the keyword "before").

The elaboration of a specification consist on the elaboration of the definitions it is composed by. Elaboration is performed by merging the binding maps corresponding to the component definitions.

- elaborate_ = Specification → action [bindings][using current bindings]

(1) elaborate() = complete

(2) elaborate⟨D_1: Definition D_2Definition$^+$⟩ =
    elaborate D_1 before elaborate D_2

2.1.2 Elaborating Definitions

Constant declarations are elaborated simply by binding the supplied token to its corresponding value.

Type declarations are elaborated by binding the token that identifies the type to a pair. The first component of this pair is a "type" semantic value. The second component is an abstraction[2] that, when enacted, performs the translation between byte streams and semantic values.

- elaborate_ = Definition → action [bindings][using current bindings]

The elaboration of a constant declaration consists on the binding of the token corresponding to the identifier I to a semantic value, corresponding to the

[2] An "abstraction" is defined in Action Notation as an item of data encoding an action. An abstraction can be stored, bound, and communicated as normal data. An abstraction can also be performed (or "enacted", following the Action Notation terminology). It corresponds more or less to the notion of procedure present in most programming languages.

valuation of the constant.

(1) elaborate ⟦"const" I: Identifier "=" C : Constant⟧ =
bind the token of I to valuation C

(2) elaborate ⟦"union" I: Identifier B: Union-body⟧ =
elaborate B before
| bind the token of I to the pair(typified)-union of B
| closure of abstraction of translate B)

2.1.3 Translating Type Declarations

The main purpose of the XDR data language is to define a standard byte-stream representation for simple and structured data. These sequences of bytes will be passed from one process to another during computation (this method ensures portability and the possibility of communication among programs written in different languages).

In order to use the XDR representation in real applications, translation algorithms need to be provided, for data types of a programming language, to and from byte representation.

A major goal in our semantic description is to obtain XDR and RPC presentations as close as possible to the actually implemented systems, without lacking mathematical rigor. This is the motivation of our choose of a "concrete" byte-stream representation of the semantic "sendable" values, instead of choosing a more abstract way.

Semantic values belonging to simple types, as well as values of structured types are translated into byte streams. In the actual C implementation, supplied with the SunOS operative system, the same routine is used to translate data in both directions. Once more, we accept the price of obscuring our semantic description to keep ourselves close to the actual implementation. Only one translation action for each type is specified. The direction of the translation depends only on the kind of data supplied to the action.

When a value is supplied to the actions defined here, they return a byte stream to represent it, following the informal specification in [10].

When a byte stream is supplied to one of the action defined here, both a value and a byte-stream are returned. The value corresponds to the beginning portion of the supplied byte-stream, interpreted as a value of the corresponding type. The returned byte stream corresponds to the "unconsumed" part of the original stream.

The functionality of the translation semantic functions can be stated as follows.

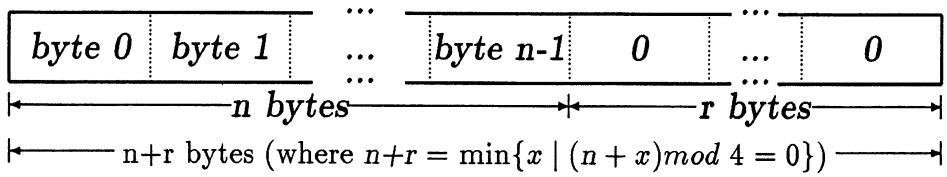
- translate_ :: Declaration → action[using a given (value | byte-stream)]
  [escaping | giving a (byte-stream |
  ⟨value, byte-stream⟩)]

The translation of an identified type, is defined as the translation action of it's definition.

(1) translate⟦Ttype-specifier Identifier⟧ = translate T

Opaque data is data that should be left uninterpreted.

The byte stream representation for fixed-length opaque data of size n, as given in [10], can be depicted as:



As it is specified in [10] the actual length of any byte representation should be a multiple of four (adding null bytes, if needed).

The translation action corresponding to a fixed length opaque data is as:

(2) translate ⟦opaque Identifier "[″V: Value"]″⟧ =

```
| | evaluate V and regive
| then
| | give the given positive-integer#1 and
| | | split the given byte-stream#2 after the given natural#1 bytes
| then
| | give the given positive-integer#1 and
| | | chop the given byte-stream#3 using the given natural#1
| then
| | give fixed-length-opaque-value of the given byte-strean #1 and
| | give the given byte-stream #2
or
| | evaluate V and
| | give component-bytes of the given fixed-length-opaque-value)
| then grow the given byte-stream #2 using the given natural #1
```

The action combinator "_ or _" makes possible the specifiction of non-deterministic actions. The combinator "_ and _" triggers the performance of its component subactions with arbitrary interleaving, while the action "A_1 then A_2" performs first the action A_1, possibly passing some transient data to the action A_2. The primitive action "regive" simply return its received transient data.

When the previous action receives a byte-stream, it is divided in two byte-streams, according to the pre-defined length of the the values of the type (given by the constant V).

The first portion will be encapsulated as an "opaque value". The second portion (possibly null) will be returned (or given) by the action.

When the previous action receives an "opaque value" the corresponding sequence of bytes is extracted and returned (after normalization).

The action "{textsfsplit _ after _ bytes" simply breaks the first argument (should be a byte sequence) in two byte sequences. The number of bytes of the first one is specified as the second argument to the function.

The actions "chop _ using _" and "grow _ using _" perform the operations of cutting and adding the zero bytes necessary to adjust the length of a byte sequence to be multiple of 4, according to a value given in the second argument.

2.2 RPC

The elaboration of a program consist on the binding of its name identifier to its program number, together with the elaboration of all its versions.

Similarly, versions and procedures elaboration bind (version and procedure) identifiers to their corresponding number in the RPC system.

(1) elaborate ⟦"program" I:Identifier "{" V : Version-def$^+$ "}"
"=" C: Constant⟧ =
| evaluate C
then
| elaborate V before bind the token of I to the given value

(2) elaborate ⟦"version" I: Identifier "{" P : program-def$^+$ "}"
"=" C: Constant⟧ =
| evaluate C
then
| elaborate P before bind the token of I to the given value

(3) elaborate ⟦T_1: Type − specifier I: Identifier "{" T_2: Type − specifier "}"
"=" C: Constant⟧ =
elaborate C before bind the token of I to the given value

2.2.1 Executing Programs

The execution of programs performing the RPC protocol is defined in this section. The same conventions as in the Sun RPC system [11] are adopted; that is: Program, version and procedure names are supposed to be in lower-case (this is ensured by our semantic functions). The existence of an "action" body is supposed for each defined procedure.

The first three semantic equations does not need explanation. The execution of a program P consist of two basic parts:

- The initialization of its environment, followed by the elaboration of itself (i.e.: the performance of all its local bindings).

- The instantiation of an Action Notation agent, whose task will be the performance of the program itself (i.e.: the performance of the abstraction that defines the program semantics)

- execute_ :: Specification → action [using current bindings | current buffer]
  [communicating]

(1) execute () = complete

(2) execute$T$: type-def = elaborate T

(3) execute$C$: Constant-def = elaborate C

(4) P = ⟦"program" I :Identifier "{" V :Version-def$^+$ "}" "=" C: Constant⟧
⇒
execute P =
| init − environment hence elaborate P
before
| subordinate an agent then
| | send a message [containing closure of abstraction of abstract P]
| | [to the given agent]

2.2.2 Abstracting Programs

This semantic function defines the action to be contracted by each agent performing an RPC service.

The main structure of the following action is an iteration. The "un-folding" combinator takes an action (possibly containing "unfold"). The whole action is equivalent to the action given by substituting itself for each occurrence of unfold in the action given as parameter.

The next action performs the first checkings for each received message (RPC call). This c ec h ks are for the correct RPC version number and the authentication of the calling process credentials, sending the corresponding error messages to the client process, if any problem is verified. In the normal case,

the processing of the message (service required) continues by the performance of the action "perform P". The complete specification of the protocol can be found in [5].

- abstract_ :: Program-def → action [using current bindings | current buffer] [communicating]

(1) $P = [\![$"program" I : Identifier "{" V : Version-def$^+$ "}" "=" C: Constant$]\!]$
⇒

```
abstract P =
unfolding
| | receive a message [from any agent][containing an rpc-msg]
 then decodify-msg
 then
| | check not (the given natural#1 is 2) and then
| | | send rpc-msg(the given xid#9, REPLY, MSG-DENIED,
| | |                RPC-MISMATCH, 2, 2)
| | | to the given agent#8
 or
| | check (the given natural#1 is 2)
 and then
| | | give the rest of the given tuple
 then regive and authenticate-call
 then
| | | check (the given boolean#9 is true) and then
| | | | send rpc-msg(the given xid#8, REPLY, MSG-DENIED,
| | | |       AUTH-ERROR, the given natural#10)
| | | | to the given agent#7
 | | | or
| | | check (the given boolean#9 is true) and then
| | | | | give the rest of the rest of the beginning
| | | |         of the given tuple
| | | | then perform P
 then unfold
```

Conclusion

This article shows that Action Notation can be use to give formal specifications of "real world" applications in a relatively simple way.

We tried to keep the semantic description as close as possible to the actual implementation. This feature imposed a very concrete style of semantic description, in which most of the actual implementation was mirrored. We believe that this relative lack of abstractness (and of mathematic beauty) of our formal specification will facilitate its use as a reference to the actual implementation.

The complete specification given in [5] is about 40 pages of formal description in which each detail of the languages semantics was Govered. Our hope is that this work can be viewed as a complementary reference to [10, 11].

Modularity of Action Notation played an important role in the present description. It permitted the specification of the two languages in a very independent fashion, in a way in which the XDR language description can be extracted as an independent entity.

References

[1] A. D. Birrell, B. J. Nelson, *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, 2(1):39-59, February 1984.

[2] P. Lee, *Realistic Compiler Generation*, The MIT Press, Foundations of Computing Series, 1989.

[3] P.D.Mosses, *Action Semantics.* Cambridge University Press, Tracts in Theoretical Computer Science Series, 1992.

[4] D. F. Brown, H. P. Moura, D. A. Watt, *Actress: an Action Semantics Directed Compiler Generatoy*, Technical report, Glasgow University, Department of Computer Science, 1992.

[5] M. Musicante, *The Action Semantics Definition of the Sun XDR and RPC Languages*, Unpublished Manuscript, Aarhus University, Department of Computer Science (Available from the author).

[6] B. J. Nelson, *Remote Procedure Call*, PhD thesis, Carnegie Mellon University, 1981a

[7] J. Palsberg, *An Automatically generated and provably correct compiler for a subset of ADA.* In Proc. ICCL'92, Fourth International Conference on Computer Languages, 1992.

[8] J. Palsberg, *A Provably Correct Compiler Generator.* In Proc. ESOP'92, European Symposium on Programming, 1992.

[9] P. G. Soares, *On Remote Procedure Call*, Area Paper, Department of Computer Science, Columbia University, January 1992.

[10] Sun Microsystems, *XDR: External Data Representation Standard.* RFC 1014, 1987.

[11] Sun Microsystems, *RPC: Remote Procedure Call Specification.* RFC 1050, 1988.

[12] D.A.Watt, *Programning Language Syntax and Semantics*, Prentice Hall International, 1991.