

SANDIA REPORT

SAND2019-xxxx

Official Use Only • Export Controlled Information

Printed February 2019

Charon Generate Parser Scripts and Their Use

Lawrence C Musson

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

OFFICIAL USE ONLY

May be exempt from public release under the Freedom of Information Act (5 U.S.C. 552), exemption number and category: 3. Statutory Exemption.

Department of Energy review required before public release

Name/Org: Lawrence C Musson / 01355 Date: September 20, 2015

Guidance (if applicable):

EXPORT CONTROLLED INFORMATION

WARNING—This document contains technical data whose export is restricted by the Atomic Energy Act of 1954, as amended 42. U.S.C. §2011. *et seq.* Violations of these export laws are subject to severe criminal penalties.

Further dissemination authorized to the Department of Energy and DOE contractors only; other requests shall be approved by the originating facility or higher DOE programmatic authority.



Sandia National Laboratories

OFFICIAL USE ONLY

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

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

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

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



SAND2019-xxxx

Official Use Only • Export Controlled Information

Printed February 2019

Charon Generate Parser Scripts and Their Use

Lawrence C Musson
Electrical Models & Simulation
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1177
lcmusso@sandia.gov

Abstract

Larry writes this

Acknowledgment

Thanks to Larry for being such an awesome dude.

Contents

1	Charon's GenerateInterpreter script	7
1.1	Charon Create Line Parser	7
1.2	Charon Create Block Parser	13
1.3	Charon Create Modifiers	14
2	Syntax Guidance	19
3	Tutorials	20
3.1	Some Quick Notes on the Interpreter	20
3.2	Creating a Simple Line Parser	21
3.3	Creating a Block Parser	26
3.4	Custom Parameter Priority	28
3.5	Anonymous Parameter Lists	29
	References	31

Figures

Tables

1 Charon's GenerateInterpreter script

The Charon interpreter is a Python script that is intended to simplify the use of Charon by providing a simpler, easier to read syntax than creating xml or yaml parameter lists. One of the most complex tasks the script must undertake is to parse the user's input and map it to the parameter lists. Or rather, one of the most complex tasks for the developer to undertake is to write the parsers that provide the mapping. This can be particularly challenging for developers who are developing routinely in C++, but rarely or never in Python.

The Charon generateInterpreter script is a script that writes scripts. Namely, it will take a simplified input provided by a developer who is required to generate new input to Charon based on a new feature or otherwise modified capability of the software and write parsers that will interpret and map user input to parameter list specifications. The Charon interpreter file can contain single lines of input or multiple lines of input collected into nested blocks. The block input is general and so nesting can go arbitrarily deep. In principle, however, the depth of nesting should be kept shallow and replaced with creativity in the input syntax. This avoids confusion for users.

The following sections describe the input file syntax for the parser generators for both line inputs and block inputs.

1.1 Charon Create Line Parser

The most fundamental element of the Charon interpreter is the line of input regardless of whether that line exists in isolation or is a part of a larger block of input. In either case, the line of interpreter input must be defined by its own parser. The parser takes the line of input and will map it to, usually, many lines of XML or YAML input in the parameter lists used by Charon. The createLineParser script in Charon's generateInterpreter scripts takes input that ultimately defines the interpreter line, all of its arguments and options including user help content and maps it to parameter list content that Charon and its libraries use. This section describes the input syntax that creates the parser for a single line of input regardless of whether it's part of a block or not.

The generateInterpreter script that creates the parser will read input files from an immediate subdirectory called parseInputs. That directory will, in general, contain inputs for all of the interpreter's parsers. If the script is called without arguments, it will sequentially process every input file in the subdirectory. Alternatively, the name of the input file can be specified as a command line argument to the script and only that input file will be processed. All of the line parser input files **must** contain a .inp suffix.

The create line parser input file contains several lines that ultimately define the parser. In general, the input file is case insensitive except in certain instances where the parameter list information is defined. Parameter lists are case sensitive. Moreover, the input file lines can be included in any order in the file, but they will be presented here in what the author considers to be a logical order.

The input to the parser generator can be grouped into two larger categories: the lines that define the interpreter input, and the lines that define the parameter list content generated by the parser. The parser that gets created is actually a Python class which the interpreter will instantiate into an object that provides mapping between user input to the interpreter and the parameter list output. The interpreter group is presented first.

The first order of business is to name the new Python class that defines the parser. It is accomplished by the line:

```
interpreter name NewParser
```

This line tells the generator to create a new Python class called

```
file
named \begin{lstlisting}charonLineParserNewParser.py
```

in the adjacent parsers subdirectory. In this instance, “interpreter” and “name” are case insensitive whereas “NewParser” is not. Case if the class and file names follow the input.

The line with the specifiers “interpreter inputLine” creates the input syntax that the interpreter will use.

```
interpreter inputLine (parse keyword) {requiredArgument} [(first option) {
    firstOptionArgument} [(second option) {secondOptionArgument}]]
```

There is no case sensitivity in this line. The initial part of this line after the specifiers tells the generator what the required parts of the line are and what the keywords are that trigger the parser. This part is

```
(parse keyword) {requiredArgument}
```

The phrase that’s contained in parentheses is the keyword that triggers the parser. The string in curly braces defines the argument to the required part of the line. There can be any number of arguments to any part of the line, but they must each be defined by a single, unspaced string. A simple example is the input of the state, or initial guess, file that will be used in a Charon job. The generator input line could be

```
interpreter inputLine (Import State File) {filename}
```

This tells the parser to look for a the keywords “import state file” and use “filename” as an argument to define parameter list entries. For example,

```
Import state file myInitialGuess.exo
```

in a Charon input file will create parameter list elements that tell Charon to look for the file “myInitialGuess.exo” as an initial guess. Import state file is not case sensitive, but the argument “myInitialGuess.exo” must be.

The parser can also be instructed to look for options on the line. This is done by enclosing optional pieces and arguments in square braces and nested as shown in the generic example. A more concise example for state files is

```
interpreter inputLine (Import State File) {filename} [(at Index) {index}]
```

This tells the parser that there may be (not required) additional instruction to use a particular time or parameter plane in the imported state file and is indicated by the optional parse keywords “at index.” By continuing with the earlier example, a line of input to the interpreter could be

```
Import state file myInitialGuess.exo at index 3
```

will tell Charon to read in a file called myInitialGuess.exo and to use the third time plane in that file as the initial guess to the simulation. In this case, “at index 3” can be omitted and certain default behaviors will be employed. Those are defined in the second group of the input file that defines the parameter list entries.

The second group of inputs all begin with an “xml” prefix regardless of whether the interpreter is ultimately instructed to generate XML formatted parameter lists or “yaml” formatted parameter lists. The one that must **always** be included are the lines that are indicated by “xmlRequired.” An example of a required line is

```
xmlRequired Charon->Mesh->Exodus File,File Name,string,{filename}
```

The xmlRequired lines will map directly from the non square bracketed parts of the interpreter input line to xml formatted parameter list. In general, a single line interpreter input may map to multiple lines of parameter lists. Each line of the parameter list that maps to required input must be separately defined and so there is no limit to the number of xmlRequired lines that can be specified. The only necessary thing is that arguments to the required interpreter input line must appear in at least one of the xmlRequired lines. In this case, it is the “filename” argument. In the xmlRequired line—and all other xml lines described later—the initial part of the line is case insensitive. Everything following that defines the parameter and how it is nested in the parameter list is case sensitive because parameter lists are case sensitive.

The format of the parameter list line follows from the requirements of Teuchos [?] parameter lists and parameters. Each parameter must have a name, type and value. These are defined in the interpreter by the comma separated section of the example above. In this case, “File Name” is the parameter name, “string” is the parameter type and “filename” is the parameter value which will be filled in by the argument value supplied by the user in the interpreter input. In other words, the xmlRequired line above will produce the following XML formatted parameter list output,

```
<ParameterList name="Charon">
  <ParameterList name="Mesh">
    <ParameterList name="Exodus File">
      <Parameter name="File Name" type="string" value="{filename}" />
    </ParameterList>
  </ParameterList>
</ParameterList>
```

The nesting of the xml output is defined by the parts of the required line that are separated by the arrow, “->” characters as seen in the filename example.

Optional inputs are very similar but the possibility of multiple options requires additional information—namely a parse keyword associated with the option. An example of an optional line that is consistent with earlier examples is,

```
xmlOptional (at Index) Charon->Mesh->Exodus File,Restart Index,int,{index}
```

The syntax in this case is identical to the required syntax except that the parse keyword “at index” has been included. Other options will have separate keywords. As in the required line, there may be any number of xmlOptional lines connected to a single option in the interpreter input. This xmlOptional line will create the XML formatted lines,

```
<ParameterList name="Charon">
  <ParameterList name="Mesh">
    <ParameterList name="Exodus File">
      <Parameter name="Restart Index" type="int" value="{index}" />
    </ParameterList>
  </ParameterList>
</ParameterList>
```

Note that this optional input will **only** be included in the parameter list if the user supplies the option. Otherwise it will not appear.

The final category in the xml input group are xmlDefault parameters. These parameters are always included in the ultimate parameter list and never contain any arguments. For example, Charon parameter list elements that contain the state guess file must also define the initial file format. For Charon, this never changes from the exodus file format and as such requiring it in input is a waste of the user’s time and serves only to make for long, confusing input prone to error. xmlDefault parameters are the cure for this. Consistent with the earlier examples,

```
xmldefault Charon->Mesh,Source,String,Exodus File
```

and tells Charon that the state file type will always be an exodus file. This line produces the XML formatted parameter list,

```
<ParameterList name="Charon">
  <ParameterList name="Mesh">
    <Parameter name="Source" type="string" value="Exodus File" />
  </ParameterList>
</ParameterList>
```

To put this all together, the createLineParser input file,

```
interpreter name ImportStateFile
```

```
interpreter inputLine (Import State File) {filename} [(at Index) {index}]
```

```
xmlRequired Charon->Mesh->Exodus File,File Name,string,{filename}

xmlOptional (at Index) Charon->Mesh->Exodus File,Restart Index,Int,{index}

xmlOptional (second Option) Charon->Mesh->Exodus File,Option Two,String,{option2}

xmldefault Charon->Mesh,Source,String,Exodus File
xmldefault Charon->Mesh->Exodus File,Restart Index,Int,-1
```

will create a parser that will take the interpreter input line

```
import state file powerMOSFET.sg.Vdrain.exo at index 7
```

that will produce the XML formatted parameter list

```
<ParameterList name="Charon">
  <ParameterList name="Mesh">
    <Parameter name="Source" type="string" value="Exodus File" />
    <ParameterList name="Exodus File">
      <Parameter name="File Name" type="string" value="powerMOSFET.sg.Vdrain.exo" />
      <Parameter name="Restart Index" type="int" value="7" />
    </ParameterList>
  </ParameterList>
</ParameterList>
```

One final comment must be made about default lines *vis-a-vis* optional lines. In this example, the “Restart Index” parameter has been included in both default and optional input. Parameter list entries are prioritized such that default entries will always be included, but will always be replaced by optional entries when they are specified. In Charon, the restart index need not always be supplied. When it isn’t, a restart index of 0 is implied. That is, the xmlDefault “Restart Index” could have been left out without penalty. It is ultimately up to the developer how explicit they feel the parameter list should be. The author prefers more explicit to less as the user won’t normally be burdened by the longer parameter list input, but its presence could be found useful at times.

Lastly, the interpreter is set up to provide help to the user when requested. For this reason, help lines are included as well that are not related to parameter lists, but only to interpreter input. The developer must always include the two entries,

```
interpreter shortHelp {Specify state file name and state index}
```

and

```
interpreter longHelp {Specify the name of the file (exodus) that contains the states
  for input/initial guess. <> Optionally, specify the index of the time plane or
  parameter plane if the file contains multiple states. <> filename is the name of
  the file. It is case sensitive. <> index is the integer index of the state to
  be used as input.}
```

edited appropriately for the input line. With the interpreter help option, the line itself will be echoed absent the () characters along with the requested verbosity. The short help line should always be succinct and in active voice. The long help line is less rigidly defined, but in general ought to be a more comprehensive description of the input and should always contain a description of the arguments in the line. The “\n” characters inserted into the long help string connote a line break for more attractive, formatted output of the help line.

Optional Priority Specification

All of the xml lines in the parser input file will have a default priority associated with it. xmlRequired and xmlOptional each default to a priority of 2. xmlDefault lines default to a priority of 1. The higher the number, the higher the priority. This is illustrated in the import state file parser with the restart index. The default value for the index is -1. If the user specifies the index in the input file to be 2, for example, the xmlOptional line that handles that option takes priority over the xmlDefault line that sets the index to -1 and so ultimately, the index is set equal to 2.

There are occasions when more than one parser generates the same line of xml parameter input that may conflict. An example of this is the ohmic BC line,

```
BC is ohmic for drain on bulk swept from 0 to 2
```

This line creates a sweep of the voltage on the drain contact to vary from 0 to 2. The default initial step size in the sweep will be 1 volt. This might be too large a step size to start with. For this reason, there is a sweep options block wherein the user can modify the behavior of the sweep. For example,

```
start sweep options
    initial step size = 0.5
end sweep options
```

This modifies the size of the initial step in the sweep. A problem arises when the sweep options block appears in the input file before the BC line. The initial step size in the sweep options block will get overridden by the step size in the BC parser. The reason for this is that the default initial step size specified in the BC parser is actually a part of the xmlOptional behavior for a sweep of a BC. So the two parameter inputs will have the same priority. The best way to handle these situations is to use an optional custom priority for the initial step size parameter in the sweep options block. It is specified thus,

```
xmlRequired Charon->Solution Control->LOCA->Step Size,Initial Step Size,double,{
    stepSize} priority 5
```

If the priority keyword does not exist in the previous line, the default priority of 2 will be assigned that parameter. By adding the priority 5 addendum to the line, one guarantees that a higher priority is assigned to this specification and will override the BC step size specification regardless of the order of appearance in the input file.

1.2 Charon Create Block Parser

In many instances, it is most convenient to organize input into blocks. The Charon interpreter allows for this, but it is encouraged that nesting of these blocks be kept to a minimum depth. The generateInterpreter script is instructed to create parsers of block input in a very similar way that line parsers are created. The difference is organizational. Block input files are read from the same parseInputs subdirectory as line parser inputs, but all contain the .blockinp suffix. An example of a material model block parser generator input is,

```
interpreterBlock name MaterialBlock

interpreterBlock (start Material Block)
```

This tells generateInterpreter to create a blockParser class called charonBlockParserMaterialBlock in a file called charonBlockParserMaterialBlock.py in the parsers directory. The keywords in the interpreter file that open the block are “start Material Block” (not case sensitive). It is implied that “end Material Block” will close the block in the interpreter file and must be provided by the user. Each block must have parameter lines inside. A separate input file that defines the parser for each of those lines as described in the previous subsection must be included. The line parser input are organized into subdirectories that mimic the nesting of the interpreter block parser. If the MaterialBlock parser in the example is defined in the .blockinp file, it is implied that there is also a subdirectory called parseInputs/MaterialBlock. This subdirectory includes all of the line parsers that are defined in the block. For example, if the material is to be named, the parseinputs/MaterialBlock directory will contain a line parser input file called createMaterialName.inp. The creation of the MaterialBlock block parser in the parsers directory will also create a parsers/MaterialBlock subdirectory that contains all line parsers associated with that first, nested level of the block. In other words, the input file parseInputs/MaterialBlock/createMaterialName.inp will create parsers/MaterialBlock/charonLineParserMaterialName.py.

It is possible for blocks of interpreter input to contain other blocks of input. That is, the parseInputs/MaterialBlock directory might contain a createDoping.blockinp file that provides for a block of line parsers in that block. Naturally then, there will be a directory called parseInputs/MaterialBlock/Doping that has in it line parser inputs that defined the parsers for each line in the doping block. To lay it out, there is a block parser input file called createMaterialBlock.blockinp,

```
interpreterBlock name MaterialBlock

interpreterBlock (start Material Block)
```

which creates a block parser invoked by the “start Material Block” keywords. There are also files in the parseInputs/MaterialBlock directory called createMaterialName.inp,

```
interpreter name MaterialName

interpreter inputLine (material name) = {materialName}

interpreter shortHelp { }
```

```
interpreter longHelp { }
```

as well as another block parser input called createDoping.blockinp,

```
interpreterBlock name Doping
```

```
interpreterBlock (start Doping)
```

This implies another subdirectory parseInputs/MaterialBlock/Doping that contains line parsers for the doping.

An interpreter input line might read something like,

```
start Material Block
  material name = Si
  start Doping
    ....
  end Doping
end Material Block
```

All parameter list input will be generated according to the individual line parsers as described in the previous section. The generated parsers are organized into the parsers directory with the same subdirectory structure as the parseInputs directory structure.

1.3 Charon Create Modifiers

The parsers created by the generate interpreter script and all associated inputs are relatively simple mappings. They translate simple, easy to understand user input to the multiple lines of nested xml formatted input that is native to Charon/Teuchos parameter lists. They contain very little logic.

It is sometimes necessary to include logic in a parser. This is possible in the Charon interpreter by using special modifier scripts, though the inputs should be designed where there is as little reliance on modifiers as possible. One example is doing parameter sweeps. The input line for a contact boundary condition may read as follows:

```
BC is contact on insulator for gate on gateoxide with work function 4.0 swept from 0
to 2
```

Clearly, the user is requesting a sweep from 0 to 2 volts on the gate contact of a FET. In the xml input for Loca, the range is specified as a minimum and maximum value and there is a signed initial step size that determines the direction of the sweep—from 0 to 2 in a positive direction in this case.

If the requested boundary condition and associated voltage sweep is slightly different, say,

BC is contact on insulator for gate on gateoxide with work function 4.0 swept from 0 to -2

the simple mapping breaks down. The minimum Loca parameter value will be set to 0 and the maximum to -2 with a sweep in the wrong direction. These special cases where a simple mapping is not possible, the interpreter allows for developer-defined modifiers to produce the correct behavior. This is the one instance where the developer must write Python code when developing parsers for the interpreter.

All of the Python code the developer must write is contained in a function titled *testForModification*. In principle, a parser may have an unlimited number of modifiers, but they should be kept to a minimum to avoid convoluting the interpreter. Any modifier will always be associated with a single, specific parser and a single file contains all modifiers that will be associated with that parser. The file must have the same canonical name as its associated parser. E.g., the parser input file for the boundary condition described here is `createContactOnInsulator.inp`. The associated files that define the modifiers must then be named `createContactOnInsulator.modifierinp.py`. All modifiers associated with this parser will be defined in the file of this name. Each modifier will be a function called `testForModification`. Each identically named function is contained inside a block that starts `start modifier X` and ends `end modifier X` where X is a unique integer number for the modifier. For example, the Python code that handles the voltage sweep input described herein is:

```
start Modifier 0

def testForModification(self,pLList):

    foundMinValue = False
    foundMaxValue = False
    foundStepSize = False
    makeMinMaxModification = False
    makeStepSizeModification = False

    for lineNumber, line in enumerate(pLList):

        # Capture the min value
        if line.find("Charon->Solution Control->LOCA->Stepper,Min Value,double") >= 0:
            lineParts = line.split(",")
            minValue = lineParts[-1]
            minValueLine = lineNumber
            foundMinValue = True

        # Capture the max value
        if line.find("Charon->Solution Control->LOCA->Stepper,Max Value,double") >= 0:
            lineParts = line.split(",")
            maxValue = lineParts[-1]
```

```

    maxValueLine = lineNumber
    foundMaxValue = True

    # Capture the initial step size
    if line.find("Charon->Solution Control->LOCA->Step Size,Initial Step Size,
        double") >= 0:
        lineParts = line.split(",")
        initialStepSize = lineParts[-1]
        initialStepSizeLine = lineNumber
        foundStepSize = True

    if foundMinValue == True and foundMaxValue == True:
        if float(minValue) > float(maxValue):
            makeMinMaxModification = True
            replacementMinLine = "Charon->Solution Control->LOCA->Stepper,Min Value,
                double,"+maxValue
            replacementMaxLine = "Charon->Solution Control->LOCA->Stepper,Max Value,
                double,"+minValue

    if makeMinMaxModification and float(initialStepSize) > 0:
        makeStepSizeModification = True
        newStepSize = str(-float(initialStepSize))
        replacementStepSizeLine = "Charon->Solution Control->LOCA->Step Size,Initial
            Step Size,double,"+newStepSize

    # Make modifications
    if makeMinMaxModification == True:
        pLList[minValueLine] = replacementMinLine
        pLList[maxValueLine] = replacementMaxLine

    if makeStepSizeModification == True:
        pLList[initialStepSizeLine] = replacementStepSizeLine

    return pLList

end Modifier 0

```

Should this parser require a second modifier, it would be contained in start modifier 1 and end modifier 1 and the function name would still be named `testForModification`. The `generateInterpreter` script will create the necessary boiler plate required to generate modifier script for the interpreter.

The modifiers will only be executed when their use is called out in the parser itself. To do this, a special line must be specified in the parser input files. The parser input file for the contact BC is

```
interpreter name ContactOnInsulator
```



```

interpreter inputLine (BC is contact on insulator for) {sidesetID} on {geometryBlock
    } with work function {workFunction} [(fixed at) {potential}[(swept from) {
    potential1} to {potential2}]]

```

```

interpreter shortHelp {Specify the potential on a contact}

```

```

interpreter longHelp {Specify the potential on a contact. <> sidesetID is the
    contact name/type <> geometryBlock is the geometry name the contact is attached
    to <> potential is the value in volts}

```

```

xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Type,string,Dirichlet
xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Sideset ID,string,{
    sidesetID}

```

```

xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Element Block ID,
    string,{geometryBlock}

```

```

xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Equation Set Name,
    string,ELECTRIC_POTENTIAL

```

```

xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Strategy,string,
    Contact On Insulator

```

```

xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS->Data,Work Function,
    double,{workFunction}

```

```

xmlOptional (fixed at) Charon->Boundary Conditions->{sidesetID}ANONYMOUS->Data,
    Voltage,double,{potential}

```

```

# Set the data parameter to a string

```

```

xmlOptional (swept from) Charon->Boundary Conditions->{sidesetID}ANONYMOUS->Data,
    Varying Voltage,string,Parameter

```

```

# Modify the solver type from NOX to LOCA

```

```

xmlOptional (swept from) Charon->Solution Control,Piro Solver,string,LOCA

```

```

#LOCA Parameters

```

```

xmlOptional (swept from) Charon->Solution Control->LOCA->Predictor,Method,string,
    Constant

```

```

xmlOptional (swept from) Charon->Solution Control->LOCA->Stepper,Continuation Method
    ,string,Natural

```

```

xmlOptional (swept from) Charon->Solution Control->LOCA->Stepper,Initial Value,
    double,{potential1}

```

```

xmlOptional (swept from) Charon->Solution Control->LOCA->Stepper,Continuation
    Parameter,string,Varying Voltage

```

```

xmlOptional (swept from) Charon->Solution Control->LOCA->Stepper,Max Steps,int,1000

```

```

xmlOptional (swept from) Charon->Solution Control->LOCA->Stepper,Max Value,double,{
    potential2}

```

```

xmlOptional (swept from) Charon->Solution Control->LOCA->Stepper,Min Value,double,{
    potential1}

```

```

xmlOptional (swept from) Charon->Solution Control->LOCA->Stepper,Compute Eigenvalues
, bool, 0
xmlOptional (swept from) Charon->Solution Control->LOCA->Step Size,Initial Step Size
, double, 1.0
xmlOptional (swept from) Charon->Solution Control->LOCA->Step Size,Aggressiveness,
double, 1.0

# Set the parameters block
xmlOptional (swept from) Charon->Active Parameters,Number of Parameter Vectors,int,1
xmlOptional (swept from) Charon->Active Parameters->Parameter Vector 0,Number,int,1
xmlOptional (swept from) Charon->Active Parameters->Parameter Vector 0,Parameter 0,
string,Varying Voltage
xmlOptional (swept from) Charon->Active Parameters->Parameter Vector 0,Initial Value
0,double,{potential1}
xmlOptional (swept from) use Modifier 0

```

This input creates a parser that specifies the voltage on the contact that is either fixed or swept from one voltage to another. The use case described here is for a voltage sweep that may be swept from high to low or low to high. The optional argument `swept from` requires the modifier to guarantee proper execution and so the line `use Modifier 0` is added to that option. This will trigger the interpreter to execute that modifier when the sweep option is selected by the user.

No modifier is executed until the input file has been completely parsed. The parameter list that is sent to the modifier script for modification will be the completed one for the current job.

2 Syntax Guidance

To set a value for a specific parameter,

```
Parameter = 1.0
```

To set a condition or model for a parameter,

```
Parameter is Vandalay cubic
```

3 Tutorials

This section contains several tutorials on how to create parsers. It starts with a tutorial on a very simple line parser with one optional section. Simple block parsers with named blocks and nested line parsers are next. Finally, advanced line parsers with custom priority and logic modifiers are last. The details of the input files to create line and block parsers are described in the initial chapter of this document. The purpose of this chapter is to take a developer step-by-step through the creation of a new parser using some of the tools available for the purpose. The developer should refer to chapter 1 for deeper insight to the creation of a parser.

3.1 Some Quick Notes on the Interpreter

In the Charon tcad-charon repository, there are two main python scripts associated with the interpreter: `charonInterpreter.py` in the `scripts/charonInterpreter` directory and `generateInterpreter.py` in the `scripts/charonInterpreter/parseGenerator` directory. The former is the user end main script and the latter the developer end main script.

The `charonInterpreter.py` script may be invoked with multiple options. The one that is almost always used is the `--input` option. This is simply the option to use a specific input file for an interpreter run. Executing

```
charonInterpreter.py --input input.inp
```

will process the contents of the `input.inp` file into the xml formatted parameter list file that is native to Charon and stops there. The filename will be `input.inp.xml`. Charon may be run through the `mpirun` command with that xml file if desired.

A Charon run may also be executed with the interpreter script with additional options. For example,

```
charonInterpreter.py --input input.inp --np 4 --run
```

This will process `input.inp` into a parameter list and then execute Charon on that parameter list on 4 processors. If the Charon executable is located in a place included in the user's `path` variable, the script will find it and use it. If not, the `CHARON_EXECUTABLE_PATH` environment variable may be set that contains the path to the executable, for example,

```
export CHARON_EXECUTABLE_PATH=/home/lcmusso/TCAD/build/src
```

There is also an implicit assumption that the name of the executable is `charon_mp.exe`. If it is not, the `CHARON_EXECUTABLE` environment variable may be set to force the interpreter to look for an otherwise named charon executable.

The `generateInterpreter.py` script is usually invoked with no options. It will automatically process all of the parser generator input files located in `scripts/charonInterpreter/parseGenerator/parseInp`

and all of its subdirectories into line and block parsers and all of the modifiers and parser libraries required by the charon interpreter. Of the of parsers will be created in the `scripts/charonInterpreter/parser` directory and will have a subdirectory structure that mirrors the directory structure under `parseInputs`.

3.2 Creating a Simple Line Parser

Whether a developer is creating a parser for a new or existing feature, the parameter list almost always antedates the interpreter parser. That is, a developer will “work the kinks out” modifying parameter list input prior even to thinking about the interpreter. So an xml formatted parameter list is generally the place to start when creating a parser to handle this part of the parameter list.

An input file to generate a parser consists of two major subsections. The first is every line that starts with the `interpreter` keyword and the second is every line that starts with the `xml*` family of keywords. `interpreter` keywords can be thought of as a front end of the interpreter. They define the name of the parser, the syntax the developer wants the user to employ and all of the help information the developer wishes the user to see. All of the keywords that start with `xml` are ones which define the parameter lists that are the target mapping of the user’s input line. Chapter 1 defines each of these in detail.

A full-blown Charon input file is not required to work on a new parser. One should always start with the parameter list entries required, create the parser generator input file and iterate until the parser reproduces the xml parameter list faithfully. There are two tools available to speed this process.

A good example of a simple parser is importing a state file. One of the first things commonly seen in a Charon parameter list is the named “initial guess” file. In interpreter parlance, this is called a state file. The xml formatted parameter list is:

```
<ParameterList name="Charon">
  <ParameterList name="Mesh">
    <Parameter name="Source" type="string" value="Exodus File" />
    <ParameterList name="Exodus File">
      <Parameter name="File Name" type="string" value="bjt2d_equ.exo" />
      <Parameter name="Restart Index" type="int" value="3" />
    </ParameterList>
  </ParameterList>
</ParameterList>
```

So what’s known about this parser is that it must contain sensible syntax to specify a file name and a restart index. And it may be desirable to make the restart index an optional part of the line.

The first thing to do is to create the input file for the parser. In this case, the file `createImportStateFile.inp` is created in the `parseInputs` subdirectory. The first line in that file gives a unique name to the parser to be created:

```
interpreter name ImportStateFile
```

In practice, it is preferred to have the name of the parser match the name of the input file with a create prefix and a .inp suffix. Note that the parser generated will be located in the parsers directory with the name `charonLineParserImportStateFile.py` with the `ImportStateFile` part of it coming from the name of the parser, not the name of the parser input file name. This keeps things systematic and easy to find.

The second line of the parser input file will contain the syntax the developer wishes the user to employ. Let's say the line should be

```
import state file bjt2d_equ.exo at index 3
```

Let's say furthermore that the only required part of this line be the specification of the state file name and that the restart index be optional. Moreover, we need to select keyword sequences which will uniquely identify this line to the parser for both the required and optional parts of this line. Keywords of the line are enclosed in parentheses and optional parts of the line are enclosed in square brackets and include a parenthetical part that contains the keywords for the option. Putting this together, the second line of the parser input file becomes

```
interpreter inputLine (Import State File) {filename} [(at Index) {index}]
```

where the keywords `interpreter inputline` tell the `generateInterpreter` script to process this line as `interpreter` syntax. The `(import state file)` section tells the interpreter the keyword sequence to trigger the parser. The `{filename}` is a variable for the name of the file the user wishes to import. It was decided that the restart index should be optional and so the section of the line that pertains to the specification of the restart index is enclosed in square brackets and necessarily contains the parenthetical `(at index)` which triggers the option processing in the parser. Note that on the interpreter end, only `filename` is case sensitive. Everything else may be any combination of upper and lower case except for `index` which must be an integer.

Two additional lines with the `interpreter` keyword contain help information for the user. In this example,

```
interpreter shortHelp {Specify state file name and state index}
```

```
interpreter longHelp {Specify the name of the file (exodus) that contains the states
    for input/initial guess. <> Optionally, specify the index of the time plane or
    parameter plane if the file contains multiple states. <> filename is the name of
    the file. It is case sensitive. <> index is the integer index of the state to
    be used as input.}
```

provide a pithy description of what this interpreter line means and a more detailed description that includes a description of optional inputs. These are invoked by the user issuing the `--help` or `--longhelp` options to the `charonInterpreter.py` script.

Now in the second major subsection of the parser input file, the parameter list and how it maps from the interpreter line must be specified so that the xml formatted input seen above is created. Internally, the interpreter does not store parameters in an xml format. It has its own format for

parameter lists in which the nesting information accompanies every parameter. Generally, the format is for example

```
Nest0->Nest1->Nest2,parameter name, parameter type, parameter value
```

The xml parameter list seen above must be translated into the interpreter format for the parser input file. For this simple example, the manual translation from xml to interpreter is relatively easy. However, when the interpreter line maps to dozens of parameters with a complicated nesting, such as might be seen for solver parameters, this process is tedious and error prone. In the `scripts/charonInterpreter/tools` directory, there is a script called `xmlToLCM.py` which does this translation automatically. This script will take a text file `xmlbjt.xml` which contains the xml formatted parameter list seen above and create a new text file called `xmlbjt.xml.LCMified` which contains the translated interpreter formatted parameter list, viz

```
Charon->Mesh,Source,string,Exodus File
Charon->Mesh->Exodus File,File Name,string,bjt2d_equ.exo
Charon->Mesh->Exodus File,Restart Index,int,3
```

These lines must be categorized as default, required and optional and added to the parser input file appropriately.

The `Source` parameter in the first line,

```
Charon->Mesh,Source,string,Exodus File
```

must always be present in an identical way. It must not ever vary the way Charon works now. It is nothing but a useless burden to require a user to put this into every input file they create. This is an ideal candidate for a default line. The interpreter will always write it into the parameter list and the user need never see it. This is specified in the parser input file as

```
xmlDefault Charon->Mesh,Source,string,Exodus File
```

The name of the file the user wishes to import must always be specified and is never a default name. This is a good candidate for a required input and is specified in the parser input file as

```
xmlRequired Charon->Mesh->Exodus File,File Name,string,{filename}
```

where the `filename` variable replaces the explicit file name in the example and ties this particular parameter to the interpreter input. That is, `{filename}` gets replaced in the parameter list by the string specified in the interpreter input file.

Lastly, we wish to include the restart index as an optional parameter. Moreover, we may also wish for the restart index to take on a default value if the option is not specified by the user. The default value for the restart index is specified in the same way the `Source` parameter was specified,

```
xmlDefault Charon->Mesh->Exodus File,Restart Index,int,-1
```

This creates a parameter `Restart Index` with a value of -1. To create the optional parameter, the parser input syntax is similar to default and required parameters, but must also contain the optional keywords,

```
xmlOptional (at index) Charon->Mesh->Exodus File,Restart Index,int,{index}
```

so the Restart Index appears twice in the parser input file. In one it contains default value information and in the other, optional user-supplied information. Here again, the {index} variable maps back to the interpreter input line.

By pulling all of this together, the entire parser input file now reads:

```
interpreter name ImportStateFile
```

```
interpreter inputLine (Import State File) {filename} [(at Index) {index}]
```

```
interpreter shortHelp {Specify state file name and state index}
```

```
interpreter longHelp {Specify the name of the file (exodus) that contains the states
    for input/initial guess. <> Optionally, specify the index of the time plane or
    parameter plane if the file contains multiple states. <> filename is the name of
    the file. It is case sensitive. <> index is the integer index of the state to
    be used as input.}
```

```
xmlRequired Charon->Mesh->Exodus File,File Name,string,{filename}
```

```
xmlOptional (at Index) Charon->Mesh->Exodus File,Restart Index,int,{index}
```

```
xmldefault Charon->Mesh,Source,string,Exodus File
```

```
xmldefault Charon->Mesh->Exodus File,Restart Index,int,-1
```

The parser is then generated by executing the generateInterpreter.py script.

The final step the developer must execute is to validate the parser by running the charon interpreter on a file that contains the new interpreter line. Say our file is bjt_2d.inp and has the line

```
import state file bjt2d_equ.exo at index 3
```

Executing

```
charonInterpreter.py --input bjt_2d.inp
```

produces a file named bjt_2d.inp.xml with the contents

```
<ParameterList name="Charon" >
  <ParameterList name="Mesh" >
    <ParameterList name="Exodus File" >
      <Parameter name="Restart Index" type="int" value="3" />
      <Parameter name="File Name" type="string" value="bjt2d_equ.exo" />
    </ParameterList>
    <Parameter name="Source" type="string" value="Exodus File" />
  </ParameterList>
```



```
</ParameterList>
```

Other than ordering of lines, this parameter list is identical. In this simple example, this is easy to see. Once again, however, if this parameter list were long and complicated, the validation will be tedious and error prone. Another script in the tools directory will make the comparison and provide a kind of diff between the two files. It reports what exists in one file that does not exist in the other. In this case, the script `compareParameterLists.py` can be used to compare `bjt_2d.inp.xml` and the file that contained the original xml input we worked from to get the following report:

```
compareParameterLists.py bjt_2dOriginal.xml bjt_2d.inp.xml
```

The following is a list of items contained in `bjt_2dOriginal.xml`, but not in `bjt_2d.inp.xml`

The following is a list of items contained in `bjt_2d.inp.xml`, but not in `bjt_2dOriginal.xml`

If something went wrong with the mapping in the interpreter and the xml generated turned out to be

```
<ParameterList name="Charon" >
  <ParameterList name="Mesh" >
    <ParameterList name="Exodus File" >
      <Parameter name="Restart Index" type="int" value="300" />
      <Parameter name="File Name" type="string" value="bjt2d_equ.exo" />
    </ParameterList>
    <Parameter name="Source" type="string" value="Exodus File" />
  </ParameterList>
</ParameterList>
```

the report would be

The following is a list of items contained in `bjt_2dOriginal.xml`, but not in `bjt_2d.inp.xml`

```
Charon->Mesh->Exodus File,Restart Index,int,3
```

The following is a list of items contained in `bjt_2d.inp.xml`, but not in `bjt_2dOriginal.xml`

```
Charon->Mesh->Exodus File,Restart Index,int,300
```

which highlights the difference. A more complicated example would be if the nesting were incorrect. The compare script is too naive to figure out how nesting should have been done. When it finds nested parameters in one file and not in another, it stops ever reporting the one contains a parameter list not contained in the other. For example, suppose instead of the parameter list Exodus

File, the parser input file contained a typo, Eoxdus File. The compare script is not smart enough to figure out the typo, but will indicate that the two files differ with the following report:

The following is a list of items contained in bjt_2dOriginal.xml, but not in bjt_2d.inp.xml

Charon->Mesh->Exodus File

The following is a list of items contained in bjt_2d.inp.xml, but not in bjt_2dOriginal.xml

Charon->Mesh->Eoxdus File

Nothing below the level of Exodus File is reported, but sufficient information is provided to the developer to indicate the error.

3.3 Creating a Block Parser

The line parsers in the Charon interpreter are where the bulk of the work is done in parsing and running a Charon job. As was described in the previous section, a single line of interpreter input can map to many lines of parameter lists. It still often makes sense to bundle line parsers into sensible groups. This makes the input file neater, more organized and easier to navigate. These groups in the interpreter are called blocks and they have their own parsers. Block parsers are in principle simpler than line parsers. Their sole purpose for existence is for organization and to make data input simpler. A good example of a block in the Charon input is where material models are spelled out for a common region of a device, e.g.

```
start Material Block siliconParameter
  material name is Silicon
  relative permittivity = 11.9
  start step junction doping
    acceptor concentration = 1e16
    donor concentration = 1e16
    junction location = 0.5
    dopant order is PN
  end step junction doping
end Material Block siliconParameter
```

In this instance, two blocks are nested. There is no limitation to the depth that blocks may be nested, but it should be limited to as few layers as makes sense to keep things simple. Block parsers control this nesting in an organized way and provide some of the mapping information.

Every line inside the blocks that begins neither with “start” nor “end” is just a line parser like the one created in the previous section’s tutorial. The way the parser input files and thus the parsers are organized mirrors the nested structure of the blocks. So the root directory for the parser

input files, `charonInterpreter/parseGenerator/parseInputs` contains the block parser for the material block. The structure and composition of the block parser input file is similar though simpler than the line parser input files. The name of the block parser input file is required only to have the suffix `blockinp`, but the following convention has been used exclusively,

```
create<BlockParserName>.blockinp
```

In this example, `BlockParserName = MaterialBlock` and the first line in the block parser makes that specification,

```
interpreterBlock name MaterialBlock
```

And just as in the line parser input file, there must be a line which specifies the keywords that trigger the block parser,

```
interpreterBlock (start Material Block) {MaterialBlockName}
```

where as before the key words are enclosed in parentheses and the variable `MaterialBlockName` is enclosed in curly braces. This represents the minimum input to create a block parser. It is often the only input.

With blocks, there is a directory structure that accompanies and mirrors the block structure. Under the `parseInputs` directory, the developer must create a subdirectory that has the same name as the block parser. In this example, it is the developer's burden to create the `parseInputs/MaterialBlock` subdirectory. Every line parser that accompanies every line in the block has an input file that lives in this subdirectory. That is, the line parsers for

```
material name is Silicon
```

and

```
relative permittivity = 11.9
```

namely `createMaterialName.inp` and `createRelativePermittivity.inp` are located in the `parseInputs/MaterialBlock` subdirectory.

This example contains one nested block layer. The directory-subdirectory structure simply repeats itself with nested blocks. There is a `createStepDoping.blockinp` block parser input file and a `StepDoping` subdirectory in the `MaterialBlock` block directory. All line parse input files related to the step doping block are located in the `StepDoping` directory.

It is important to note that very commonly, parameters created by line parsers in blocks almost always need to know the block name in order to get the parameter list correct. Every line parser inside the blocks will have defined any variable specified in the block parser. In this example, the variable `MaterialBlockName` is defined and can be used in every line parser input file that exists under the `MaterialBlock` directory tree. The `MaterialName` line parser input file is

```
interpreter name MaterialName
```

```

interpreter inputLine (material name) is {materialName}

interpreter shortHelp {Set the Material name }

interpreter longHelp {Set the Material name <> materialName is the material in this
    material block (e.g. Silicon)}

xmlRequired Charon->Closure Models->{MaterialBlockName},Material Name,string,{
    materialName}

```

3.4 Custom Parameter Priority

It can happen on occasion that a parameter gets defined more than once. Most of the time, this presents no problem. But sometimes it does. An example of this is doing potential sweeps on contacts. An interpreter input line for sweeping is

```
BC is ohmic for anode on silicon swept from 0.5 to 0.25
```

The parser that processes this line will create the appropriate parameter list for the boundary condition. It also creates as xmlOptional parameters the necessary inputs to switch the run from NOX solver to LOCA solver and sets LOCA stepper information. This parser will set the initial step size of the sweep to 1.0. A step size of 1 volt is often far too large to take and so a smaller step size should be specified. This can be done in an optional block for sweep parameters,

```

start sweep options
    initial step size = 0.25
end sweep options

```

With this, a parameter has been set that creates an initial step size of 1.0 and a second one has been set that creates an initial step size of 0.25. This presents a potential problem because in the first instance, the parameter is xmlOptional and the second, the parameter is xmlRequired. Optionals and Requireds have the same priority and the one which will get used is the one last encountered in the list. This makes the input file order dependent and undesirable. This is remedied by adding a custom priority tag to the desired specification. When the user adds the optional sweep options block, anything specified in there must take priority over anything else that is default behavior. The tag is set in the line parser input file for the initial step size in the sweep option block as follows:

```

interpreter name InitialStepSize

interpreter inputLine (Initial Step Size) = {stepSize}

interpreter shortHelp {Specify initial step size for parameter sweep}

interpreter longHelp {Specify initial step size for parameter sweep <> {stepSize} =
    the step size}

```

```
xmlRequired Charon->Solution Control->LOCA->Step Size,Initial Step Size,double,{
    stepSize} priority 5
```

If the priority tag is missing from the xmlRequired line, default priority is assigned. If the tag is present, the requested priority is assigned.

3.5 Anonymous Parameter Lists

There are unfortunate instances in the Charon parameter lists that a parameter list is unnamed. This is called an anonymous parameter list and it does create an organizational hazard. The interpreter relies on a unique nesting path to do its mapping from interpreter specifications into parameter lists. These are managed in a special way in the interpreter and some care must be taken to get them right. The parameter list name ANONYMOUS is always used where there is an unnamed parameter list. If the unnamed parameter list has no peers, that is if it sits alone at its particular nesting level and path, ANONYMOUS is all that is required to handle the mapping correctly. However, if there is more than one, ANONYMOUS is not unique and the mapping will not be done correctly. An important example of this is contact boundary conditions. The boundary conditions on a diode are as follows:

```
<ParameterList name="Charon">
  <ParameterList name="Boundary Conditions">
    <ParameterList>
      <Parameter name="Type" type="string" value="Dirichlet"/>
      <Parameter name="Sideset ID" type="string" value="anode"/>
      <Parameter name="Element Block ID" type="string" value="silicon"/>
      <Parameter name="Equation Set Name" type="string" value="ALL_DOFS"/>
      <Parameter name="Strategy" type="string" value="Ohmic Contact"/>
      <ParameterList name="Data">
        <Parameter name="Voltage" type="double" value="0.5"/>
      </ParameterList>
    </ParameterList>
    <ParameterList>
      <Parameter name="Type" type="string" value="Dirichlet"/>
      <Parameter name="Sideset ID" type="string" value="cathode"/>
      <Parameter name="Element Block ID" type="string" value="silicon"/>
      <Parameter name="Equation Set Name" type="string" value="ALL_DOFS"/>
      <Parameter name="Strategy" type="string" value="Ohmic Contact"/>
      <ParameterList name="Data">
        <Parameter name="Voltage" type="double" value="0"/>
      </ParameterList>
    </ParameterList>
  </ParameterList>
</ParameterList>
```

The anode and cathode are peer parameter lists and are unnamed and are not unique. Internally, Trilinos-Teuchos will replace anonymous parameter lists with assigned names child0, child1

etc., but the user never sees it. The interpreter can only handle parameters with a unique path. The way the interpreter manages this is that any parameter list named ANONYMOUS will get written to the mapped parameter list as unnamed. This is true even if ANONYMOUS represents only a fraction of the name. The parser input file in part for boundary conditions is

```
interpreter name OhmicBC
```

```
interpreter inputLine (BC is ohmic for) {sidesetID} on {geometryBlock} [(fixed at) {
    potential}[ (swept from) {potential1} to {potential2}]]
```

```
interpreter shortHelp {Specify the potential on a contact}
```

```
interpreter longHelp {Specify the potential on a contact. <> sidesetID is the
    contact name/type <> geometryBlock is the geometry name the contact is attached
    to <> potential is the value in volts}
```

```
xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Type,string,Dirichlet
```

```
xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Sideset ID,string,{
    sidesetID}
```

```
xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Element Block ID,
    string,{geometryBlock}
```

```
xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Equation Set Name,
    string,ALL_DOFS
```

```
xmlRequired Charon->Boundary Conditions->{sidesetID}ANONYMOUS,Strategy,string,Ohmic
    Contact
```

Because the sidesetID that must be specified for the boundary condition will be unique for that parameter list, it is prepended to the ANONYMOUS part of the parameter. This has the effect of giving the interpreter a unique name to work with. When the parameter list is ultimately mapped and written, the entire sidesetIDANONYMOUS string gets replaced with nothing and becomes an unnamed parameter list.

References

- [1] *Sam Myers Personal Communications.*
- [2] Gary L. Hennigan, Robert J. Hoekstra, Joseph P. Castro, Deborah A. Fixel, and John N. Shadid. Simulation of neutron radiation damage in silicon semiconductor devices. Technical report SAND2007-7157, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 2007.
- [3] Eric R Keiter, Ting Mei, Thomas V Russo, Eric Lamont Rankin, Richard Louis Schiek, Keith R Santarelli, Heidi K Thornquist, Deborah A Fixel, Todd S Coffey, and Roger P Pawlowski. Xyce parallel electronic simulator : users' guide, version 4.1. Technical Report SAND2008-6461, Albuquerque, New Mexico 87185, 2009.
- [4] Eric R Keiter, Ting Mei, Thomas V Russo, Richard L Schiek, Peter E Sholander, Heidi K Thornquist, Jason C Verley, and David G Baur. Xyce parallel electronic simulator users' guide, version 6.3. Technical report SAND2015-3377, Sandia National Laboratories, Albuquerque, New Mexico 87185, April 2015.
- [5] Kevin M Kramer and W Nicholas G Hitchon. *Semiconductor Devices: A Simulation Approach*. Prentice Hall PTR, 1997.
- [6] Lawrence C Musson. Modeling of cluster defects in xyce: Theory and user guide. Technical report SAND15-9110, Sandia National Laboratories, Albuquerque, New Mexico 87185, 2015.
- [7] Lawrence C Musson, Gary L Hennigan, Xujiao Gao, Andy Huang, and Mihai Negoita. Charon user manual. Technical report SAND19-xxx, Sandia National Laboratories, Albuquerque, New Mexico 87185, 2019.
- [8] Samuel M. Myers, P. J. Cooper, and William R. Wampler. Model of defect reactions and the influence of clustering in pulse-neutron-irradiated si. *Journal of Applied Physics*, 104, 2008.
- [9] S.M. Myers, P.J. Cooper, and W.R. Wampler. Model of defect reactions and the influence of clustering in pulse-neutron-irradiated si. *J. Appl. Phys.*, 104, 2008.
- [10] Rolf Riesen. How to be conformant. *Psychology Today and Tomorrow*, 784(3):121–130, 2002.
- [11] M.T. Robinson and I.M. Torrens. Computer simulation of atomic-displacement cascades in solids in the binary-collision approximation. *Phys. Rev. B*, 9, 1974.
- [12] Donald Shepard. A two-dimensional interpolation function for irregularly spaced data. *Proceedings 1968 ACM National Conference*, pages 517–524, 1968.
- [13] John E. Smith. On the use of dry erase markers in science. *Journal of Pen and Pencil*, 784(3):121–130, 2002.

- [14] William R. Wampler and Samuel M. Myers. Model for transport and reaction of defects and carriers within displacement cascades in gallium arsenide. *Journal of Applied Physics*, 117, 2015.

OFFICIAL USE ONLY



Sandia National Laboratories

OFFICIAL USE ONLY