Table of Contents

1. Running sigil

Sigil toolchain is a tool for dynamically profiling computation and communication costs. To run the sigil toolchain, we need to run sigil, callgrind and postprocessing. We explain this as two steps processes, running sigil and post processing. Each step can be done manually or using scripts. We first explain the manual version and then the scripts.

2. Running sigil manually

To run sigil manually, there are two steps; 1. Run sigil tool chain and 2. Run post processing scripts.

1. Compile your source code.
2. Run sigil tool chain by following command:

The typical usage of Sigil is as follows:

<valgrind build location>/vg-in-place --tool=callgrind <Callgrind command line options> --sigil-tool=yes --separate-callers=<X> --cache-sim=yes --branch-sim=yes --drw-func=yes <location to user program/executional user program> [1]

The options are necessary to run Sigil. Table 1 shows sigil options and table 2 will show callgrind options. These options are necessary and must be included.

Table1. Sigil options

Sigil options	
Options	Description
--sigil-tool= “yes/no”	Default is no (i.e. run stock Callgrind)
--drw-func= ”yes/no”	Enable function-level communication profiling
--drw-syscalltracing= ”yes/no”	Enabling tracing of syscalls dependencies during

	profiling.
--drw-smlimit= ”yes/no”	Limit on the size of shadow memory (in MB)
--drw-debug= ”yes/no”	If “yes”, prints memory usage information that can be used for debugging.
--drw-events	Capturing events allows reconstruction of dependencies over the program run. Costs significantly more memory during profiling. Default is "no".
--drw-datareuse	If “Yes” enables examining of byte-level reuse during profiling. Default is "no".

This will give you an output called “sigil.totals.out-1. Please refer to test folder to see more examples.

In addition to to sigil output we will need callgrind output; you can obtain this file by running : <location to sigil core>/valgrind-3.7/vg-in-place --tool=callgrind --cache-sim=yes --branch-sim=yes --callgrind-out- file=<desired location for outputfile>/<callgrind_output_file> <desired location for output file>/<executable file>

Table 21. Callgrind options

Callgrind options	
Options	Description
--tool= callgrind	Tells Valgrind to run the callgrind tool.
--cache-sim= ”yes”	Tells callgrind to run a cache simulation
--branch-sim= ”yes/no”	Branch prediction simulation
--separate-callers=<number of callers> [2]	large enough to support the branch of a calltree with the largest depth

The highlighted fields need to be reconfigured with your environmental variables. Sigil_path is where sigil directory has been setup. This command will use callgrind tool and renames the output file with << callgrind_out>>. The options, cache-sim, branch-sim are necessary for post processing script. Callgrind_output_file and exe_file_path are where you want to place callgrind and the test executable file.

Running the post processing script on sigil and callgrind tool chains’ output: postprocessing_path/aggregate_costs_gran.py $exe_file_path/sigil.totals.out-1 --trim-tree --cgfile=$callgrind_output_file/callgrind_out --gran-mode=metric > $callgrind_output_file/postprocessing_result_test

For running Post processing script, first sigil and callgrind need to run; the aggregate_cost.py generates a control data flow graph representation using Sigil and callgrind output, runs a partitioning flow to trim the tree the leaves of the resulting tree are best candidates for accelerators. Table 3 summarizes data fields extracted from both sigil and callgrind outputs.

Table 3. Sigil post processing output

Output field	Descriptions
Numcalls	number of calls to that node
Instructions	number of instructions executed in that node
Flops	Total Floating Point operations executed in that node
Iops	Total Integer operations executed in that node
Ipcomm_uq	The unique input communication seen by that node (inclusive).
Opcomm_uq	The unique output communication seen by that node (inclusive).
Local_uq	The unique local communication seen by that node (inclusive).
Ipcomm	The total input communication seen by that node (inclusive).
Opcomm	The total output communication seen by that node (inclusive).
Local	- The total local communication seen by that node

	(inclusive).
Comp_Comm_uniq	The ratio of computation to unique communication. This represents the work done per byte of true input data transferred. Higher ratios are more indicative of a hotspot.

There are more options that can be used, please run the post-processing script with --help to see the options and their description.

Note that the result of these files can increase fast in a very short amount of time, especially if the program has many calls to a series of functions such as biological neural network simulations. Table 4 shows this case for Izhikevich model[1] some examples to that effect:

Table 4 post-processing file size and time relation

Program	Description	Time	Size of post processing file
Izhikevich model C code	Computation with spikes (spiking neural network code)	10s	38.6 Kb
`Hund-Rudy Dynamic`simulation	Cellular simulation	10s	2.6 Mb

3. Sigil Script Description

3.1. Sigil Scripts Flow

Figure 2 shows the flow of scripts in this document. Sigil_presetup.sh, downloads the pre-requisite packages; sigil_setup.sh sets up sigil tool chain and post processing. At the end sigil_test.sh is presented for aggregation costs .

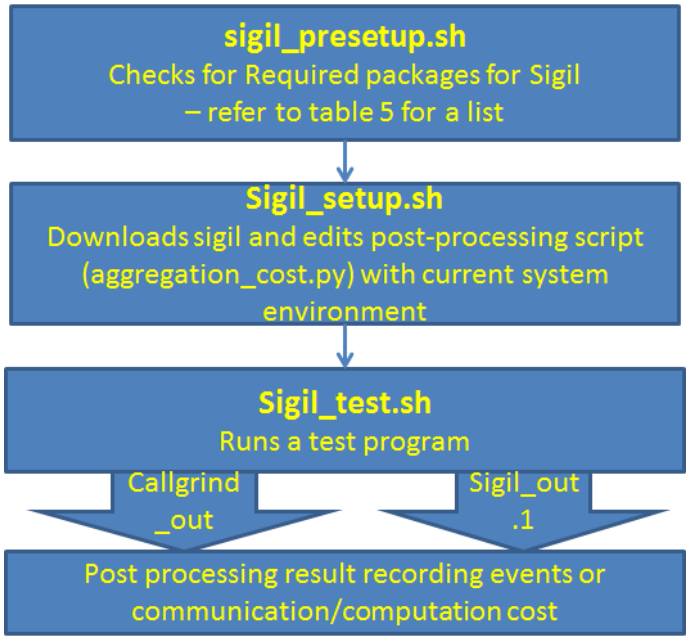

Figure 1 Sigil general Script Flow

3.2. Sigil make script

The script sets up sigil by asking the user for preferred path for sigil, it also shows the default path. Once sigil is cloned, it makes the sigil tool chain and configures “aggregate_costs_gran.py” with sigil_path. Here, we briefly explain the scripts.

This block of code asks user to insert their code or choose the default path. The default path will be displayed on screen at runtime.

```
unset USER_DIR
export sigil_path=~/sigil

line_header=">>>>>"
read -p "$line_header Specify the Sigil install directory [$sigil_path]:" USER_DIR

if ! [ -z "$USER_DIR" ]
then
  sigil_path=$USER_DIR
fi

echo $sigil_path
mkdir $sigil_path
export sigil_path=$sigil_script
```

At this point of script the sigil code is downloaded and sigil tool chain will be set up.

```
cd $sigil_path

#downloading sigil
git clone dragon:/archgroup/projects/sigil.git

#making sigil
cd $sigil_path/Sigil/valgrind-3.7/

./autogen.sh
./configure
make
```

To use the post script aggregate_cost_gran that provides the communication and computation cost of nodes, you will need to edit the file. We will do this by finding the pattern and changing it to the user’s path. The postprocessing_path in sigil_test points to regression analyzing script called aggregate_costs_gran.py which is located in <sigil/postprocessing> folder. This script uses callgrind parameters to calculate number of instructions (Integer and floating point) and number of calls to a node. These parameters are explained in table 3.

```
#make sure you add the scripts path to your command lines when running sigil
#this part will setup the post processing scripts

#this is for sed added the bash and aggregate_costs_gran
postprocessing_path="$postprocessing_path $sigil_path/Sigil/postprocessing/aggregate_costs_gran.py"
export postprocessing_path

#where sigil's callgrind_annotate which is in valgrind-3.7_original/callgrind/callgrind_annotate
callgrind_annotate_path="$callgrind_annotate_path $sigil_path/Sigil/valgrind-3.7/callgrind"
export callgrind_annotate_path

#change the path to where your post-processing script is
callgrind_annotate_new='cg_anno = "perl $sigil_path/valgrind-3.7/callgrind/callgrind_annotate --
threshold=100 " + callgrind_filename'
callgrind_annotate_inclusive_new='cg_anno = "perl $sigil_path/valgrind-
3.7/callgrind/callgrind_annotate --inclusive=yes --threshold=100 " + callgrind_filename'


#don't change anything from here it is what it was in old script
callgrind_annotate_old='cg_anno = "perl /archgroup/archtools/Profilers/valgrind-
3.7_original/callgrind/callgrind_annotate --threshold=100 " + callgrind_filename'
callgrind_annotate_inclusive_old='cg_anno = "perl /archgroup/archtools/Profilers/valgrind-
3.7_original/callgrind/callgrind_annotate --inclusive=yes --threshold=100 " + callgrind_filename'
```

```
#echo callgrind_annotate_new

sed -i 's|'"$callgrind_annotate_old"'|'"$callgrind_annotate_new"'|' $postprocessing_path
sed -i 's|'"$callgrind_annotate_inclusive_old"'|'"$callgrind_annotate_inclusive_new"'|'
$postprocessing_path
```

3.3 Examples of sigil usage

We will show sigil results of two programs a math function and a a biological cell simulation. Our point is to show hot functions in each of these programs.

3.3.1 Mathematics Function

Consider the following code

```
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}
int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return r;
}

int mult (int a, int b)
{
  int r;
  r=a*b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  z = subtraction (8,3);
  cout << "The result is " << z;
  z = mult (7,8);
  cout << "The result is " << z;
}
```

./sigil_test.sh will run sigil on an executable file.“ The variables are explained in Table 5, while the lines are explained in Table 6. The following script creates a file postprocessing_result. In this document we only explain the commands in the script that will compile a c code and run sigil on the executable.

```
cd
export exe_file_path="$exe_file_path $sigil_path/sigil/sigil_test"
cd $exe_file_path
g++ $exe_file_path/sigil_test.c -o $exe_file_path/test_math
```

The following line will provide sigil output. It first goes to execution_path , compiles the test program with g++ and then runs sigil and callgrind on it. The output of these two will be fed to postprocessing script and will provide postprocessing_result_math.

```
$sigil_path/sigil/valgrind-3.7/vg-in-place --tool=callgrind --sigil-tool=yes --separate-callers=100
--cache-sim=yes --drw-func=yes $exe_file_path/test_math #--drw-events=yes,
```

```
$sigil_path/sigil/valgrind-3.7/vg-in-place --tool=callgrind --cache-sim=yes --branch-sim=yes --
callgrind-out-file=callgrind_out $exe_file_path/test_math

$sigil_path/sigil/postprocessing/aggregate_costs_gran.py sigil.totals.out-1 --trim-tree --
cgfile=callgrind_out --gran-mode=metric > $exe_file_path/postprocessing_result_math
```

Note the red rectangle signifies numbers that represent either operating system functions or those that are not included in binary. The results have been explained in table 3 which is repeated here for convenience.

The first part sets the environmental variables, table 6 lists these environments.

Table 5 environmental variables that need to be set

Variable	Description
sigil_path	Where sigil has been compiled
exe_file_path	Where user's executable file (after compiling has been stored)
Postprocessing_path	Where ./aggregate_costs_gran.py is located
callgrind_output_file	Where user stores the output of callgrind

Table 6 shows the commands and their explanations.

Table 6 main commands for using sigil

Command line	Explanation
$sigil_path/vg-in-place --tool=callgrind --sigil-tool=yes --separate-callers=100 --cache-sim=yes --drw-func=yes $exe_file_path/math	sigil command with necessary options, the minimum of which is 100. Cache-sim is a fag that "enables or disables collection of cache access and miss counts"[1]. exe_file_path/math is user's executable file.
$sigil_path/vg-in-place --tool=callgrind --cache-sim=yes --branch-sim=yes --callgrind-out-file=$callgrind_output_file/callgrind_out $exe_file_path/math	callgrind runnable with necessary options. The out_put is renamed to callgrind_out [1].
postprocessing_path/aggregate_costs_gran.py $exe_file_path/sigil.totals.out-1 --trim-tree --cgfile=$callgrind_output_file/callgrind_out --gran-mode=metric > $callgrind_output_file/postprocessing_result	aggregate_costs_gran.py is the script that calculates aggregate costs as shown in table 3 and 4. It uses sigil and callgrind output and provides postprocessing_results.

The complete test result is included in sigil folder. The nodes in the bottom are the candidates for acceleration. As expected addition, subtraction and multiplication are candidates for accelerations.

Table 3. Sigil post processing output

Output field	Descriptions
Numcalls	number of calls to that node
Instructions	number of instructions executed in that node
Flops	Total Floating Point operations executed in that node
Iops	Total Integer operations executed in that node
Ipcomm_uq	The unique input communication seen by that node (inclusive).
Opcomm_uq	The unique output communication seen by that node (inclusive).
Local_uq	The unique local communication seen by that node (inclusive).
Ipcomm	The total input communication seen by that node

	(inclusive).
Opcomm	The total output communication seen by that node (inclusive).
Local	- The total local communication seen by that node (inclusive).
Comp_Comm_uniq	The ratio of computation to unique communication. This represents the work done per byte of true input data transferred. Higher ratios are more indicative of a hotspot.

You can use this script after you have compiled your file. Sigil takes the executable file, sigil toolchain assigns shadow memory to interact communication and computation costs then post processing script extract information that can be used for hardware-software design process. Figure 3 shows the acceleration candidates, while figure4 shows these functions values.

```
Trimming tree.....
Func num   Func Inst  Function name                                  Numcalls   Instrs        Flops         Iops          Ipcomm_uq  Opcomm_uq  Local_uq   Ipcomm     Opcomm

Uppertree Software Time (Cycles): 408249.000000
Bottom nodes Software Time (Cycles): 7470.000000

424        0          std::ctype<char>::_M_widen_init() const        * 1        2776          0             6513          136        16         670        141        16
434        0          0x0000003cb9284100                             * 3        75            0             105           24         2          0          24         8
390        0          addition(int, int)                             * 1        10            0             20            8          0          20         8          0
408        0          _IO_doallocbuf                                 * 1        113           0             153           32         44         52         32         44
415        0          _IO_default_xsputn                             * 1        713           0             1214          44         54         168        368        54
437        0          subtraction(int, int)                          * 1        11            0             22            8          0          20         8          0
438        0          mult(int, int)                                 * 1        10            0             20            8          0          20         8          0
```

Figure 3. partial sigil post processing output for simple_math

	Function name	Numcalls	Instrs	Flops	Iops	Ipcomm_uq	Opcomm_uq	Local_uq	Ipcomm	Opcomm
396	0	main	* 1	15870	0	26786	1405	414	4783	1543	452
429	0	std::ostream::operator<<(int)	* 3	7053	0	12865	871	295	2147	947	323
432	0	std::ostream& std::ostream::_M_insert<long>(long)	* 3	6209	0	11522	779	295	1793	855	323
435	0	std::ctype<char>::_M_widen_init() const	* 1	2627	0	6250	296	35	651	308	35
436	0	std::ctype<char>::do_widen(char const*, char const	* 1	1316	0	2132	536	32	371	548	32
175	28	_dl_runtime_resolve	* 1	1190	0	1958	272	32	355	284	32
176	28	_dl_fixup	* 1	1169	0	1940	264	32	299	276	32
68	29	_dl_lookup_symbol_x	* 1	1089	0	1811	240	36	263	268	36
69	29	do_lookup_x	* 1	943	0	1612	235	12	171	239	12
38	14	_dl_name_match_p	* 5	258	0	411	63	0	16	87	0
22	47	strcmp	* 10	133	0	246	13	0	0	85	0
70	29	check_match.12439	* 1	196	0	392	67	0	38	67	0
22	48	strcmp	* 2	137	0	270	16	0	0	16	0
439	0	memcpy	* 1	113	0	165	264	0	8	264	0
441	0	std::num_put<char, std::ostreambuf_iterator<char,	* 3	2440	0	3560	385	232	624	451	260
175	29	_dl_runtime_resolve	* 1	1510	0	2284	164	71	329	166	85
176	29	_dl_fixup	* 1	1489	0	2266	156	68	273	158	76
68	30	_dl_lookup_symbol_x	* 1	1409	0	2137	132	64	237	150	64
69	30	do_lookup_x	* 1	408	0	703	127	40	133	129	40
70	30	check_match.12439	* 1	137	0	278	48	0	33	48	0
22	49	strcmp	* 1	86	0	170	8	0	0	8	0
444	0	std::ostreambuf_iterator<char, std::char_traits<ch	* 3	896	0	1232	223	164	292	293	188
413	1	__gnu_cxx::stdio_sync_filebuf<char, std::char_trai	* 3	458	0	620	160	91	104	164	91
416	1	fwrite	* 3	440	0	617	136	91	104	140	91
417	1	_IO_file_xsputn@@GLIBC_2.2.5	* 3	245	0	302	48	43	48	68	43
448	0	0x00000000007d680	* 3	75	0	105	24	4	0	24	8
445	0	0x0000000000080920	* 3	87	0	81	48	0	42	128	0
248	14	std::locale::id::_M_id() const	* 3	15	0	18	16	0	0	48	0
175	27	_dl_runtime_resolve	* 1	887	0	1395	163	73	338	163	75
176	27	_dl_fixup	* 1	866	0	1377	155	73	282	155	75

Figure4. Partial output for test program

3.3.2 Hund-Rudy Dynamic (HRd) Model

Sigil can be used to in hardware/software partitioning process. Figure 4 is the show case of this simulation with the inputs and outputs, mainly currents values. It can be used on all sorts of applications. Here we look into a biological application. Heart arrhythmia causes hundred thousand deaths in U.S. annually. Pacemakers are small electronic device that is place in patents body and provides a low energy electrical pulse to stimulate heart to beat in a normal rate. However, common pacemakers are in need of

constant monitoring and maintenance. Here we look at a code that simulates biological cells. This specific model, simulates heart cells that are believed to be an important component of cardiac physiological state, therefore, it is crucial in heart arrhythmia detection [2]. These types of simulation are being proposed to design biological, automatic responsive pacemakers to replace hundreds of thousands static hardware pacemakers that are currently being used [3]. The following model provides information about tissue mechanics, metabolism and blood flow at larger scales, therefore it has a lot of classes and their specific functions [2]. Table 8 shows part of sigil post-processing result for this code [4]. Further material and the code can be seen in references[5].

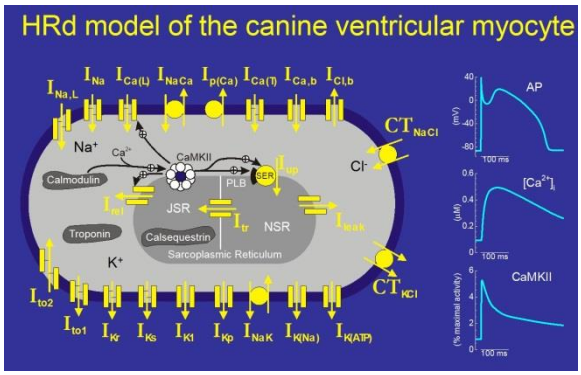

Figure 4 HRd model

Table 8 Sigil post-processing output for main classes of Hund-Rudy Dynamics.

Numcalls	description	Instructions	FLOPS	IOPS	Ipcomm_uq	Opcomm_uq	Local_uq	Ipcomm	Opcpmm	Local	Comp_Comm_uniq
Main	Calls all the functions	1	6537990898	9188921	7857328039	1935832	216	1230751211	37951699	343	6554052080
Cell_data	contains the set of dynamic variables that define the 'state' of an HRD cell at a given time	241708	85408361	483416	205470321	9474244	13501503	29938029	17250138	24169183	121191995
cell::currents(timer)	provides functions for determining the next time step and implementing stimulus protocols	241708	5287792102	8705337	6402883801	90894771	27078159	951072193	261486046	63835832	5015434105
HRDinaca::computelnaca	contains constant-valued parameters that are used throughout the model	241708	999807443	139	1219798002	11705056	5811860	112405964	50384296	7774780	850258212

```
Trimming tree.....
Func num   Func Inst  Function name                                  Numcalls   Instrs        Flops        Iops          Ipcomm_uq  Opcomm_uq  Local_uq   Ipcomm     Opcomm

Uppertree Software Time (Cycles): 409158.000000
Bottom nodes Software Time (Cycles): 2437817005.000000

568        3          pow                                        * 6802040    2324232775    324          2951069349    26870156   16339880   185765046  64976300   16361368
```

Figure 5. Acceleration candidate for HRd simulation

		Function name	Numcalls	Instrs	Flops	Iops	Ipcomm_uq	Opcomm_uq	Local_uq	Ipcomm	Opcomm
389	0	main	* 1	6537990898	9188921	7857328039	1935832	216	1230751211	37951699	343
555	0	cell::currents(timer)	* 241708	5287792102	8705337	6402883801	90894771	27078159	951072193	261486046	63835832
582	0	HRDinaca::computeInaca()	* 241708	999807443	139	1219798002	11705056	5811860	112405964	50384296	7774780
568	3	pow	* 2417080	825360601	104	1047275793	7835896	5810880	65979700	21373032	5839172
569	3	__ieee754_pow	* 2417080	627160041	104	667794233	34808672	9888	27306420	34810160	38180
570	3	__exp1	* 2175439	193613656	0	145753500	1933792	0	1933792	17403512	0
627	2	__slowpow	* 16	4021462	104	10096358	2848	9888	234676	4336	38180
629	2	__mplog	* 16	2645399	64	6647034	5128	11940	155272	25044	51044
609	12	__mpexp	* 32	2583058	64	6493782	3964	8196	142464	20680	36756
614	48	__mul	* 1504	1615766	0	4091904	18256	15424	22540	145568	63936
615	12	__dvd	* 96	896181	0	2237472	28288	9024	103160	239616	9024
616	12	__inv	* 96	792096	0	1976928	18816	8832	88888	181632	57216
614	49	__mul	* 576	632224	0	1604544	39748	57984	53204	691968	106368
609	13	__mpexp	* 16	1336816	32	3355636	4944	6152	73056	21152	11444
614	53	__mul	* 772	855715	0	2161828	9056	7648	12032	72944	31904
615	13	__dvd	* 48	448129	0	1118736	14544	4344	52304	120208	4344
616	13	__inv	* 48	396048	0	988464	9408	4440	45232	90816	28632
614	54	__mul	* 288	316112	0	802272	20796	28992	27136	345984	53184
554	11	isnan	* 4834160	67678240	0	159527280	19336640	0	19336640	38673280	0
561	10	finite	* 2417080	16919560	0	38673280	19336640	19336640	0	19336640	19336640
559	7	exp	* 725124	101692734	35	138924797	7736512	1934668	17421256	9674720	1935640
560	7	__ieee754_exp	* 725124	77038518	35	93241985	11603840	1004	5819272	11608384	1976
561	11	finite	* 725124	5075868	0	11601984	5800992	3867328	0	5800992	3867328
571	0	HRDical::computeIcal(double)	* 241708	798882799	332	944068074	16552467	9688288	97748696	47519383	17479290
559	1	exp	* 4109036	576797888	309	788551256	3883363	1950804	27225112	34851395	1998900
560	1	__ieee754_exp	* 4109036	437090664	309	529681988	48357635	17140	21424120	48387043	65236
607	2	__slowexp	* 53	1533007	309	3725592	16387	17140	153816	45867	65236
609	4	__mpexp	* 53	1453957	72	3548545	16896	28660	137032	58376	87716
614	16	__mul	* 1362	757998	0	1890822	19284	17080	26772	110580	48468
615	4	__dvd	* 159	611542	0	1471545	32308	9776	90152	137416	9872
616	4	__inv	* 159	514577	0	1236861	21220	9636	71664	94316	41532
614	17	__mul	* 636	359552	0	899092	35788	43672	39420	337064	75472

Figure 6 Partial result of HRd model sigil post-processing

As it is shown in figure 5, after 5 hours of simulation, based on our aggression costs definition, it is just power function that is worth acceleration. Also figure 6, shows cell::currents(timer) and HRDinaca::computeInaca are both called by main.
Also, mul (multiplication) and dvd (divide) have been called by two different classes and you can see the instances of each function.

References

[1] R. D. Stewart and W. Bair, "Spiking neural network simulation: numerical integration with the Parker-Sochacki method," *J. Comput. Neurosci.*, vol. 27, no. 1, pp. 115–133, Aug. 2009.

[2] R. H. Clayton, O. Bernus, E. M. Cherry, H. Dierckx, F. H. Fenton, L. Mirabella, A. V. Panfilov, F. B. Sachse, G. Seemann, and H. Zhang, "Models of cardiac tissue electrophysiology: Progress, challenges and open questions," *Prog. Biophys. Mol. Biol.*, vol. 104, no. 1–3, pp. 22–48, Jan. 2011.

[3] R. B. Robinson, "Engineering a biological pacemaker: in vivo, in vitro and in silico models," *Drug Discov. Today Dis. Models*, vol. 6, no. 3, pp. 93–98, 2009.

[4] F. Fenton and E. Cherry, "Models of cardiac cell," *Scholarpedia*, vol. 3, no. 8, p. 1868, 2008.

[5] T. J. Hund and Y. Rudy, "Rate Dependence and Regulation of Action Potential and Calcium Transient in a Canine Cardiac Ventricular Cell Model," *Circulation*, vol. 110, no. 20, pp. 3168–3174, Nov. 2004.