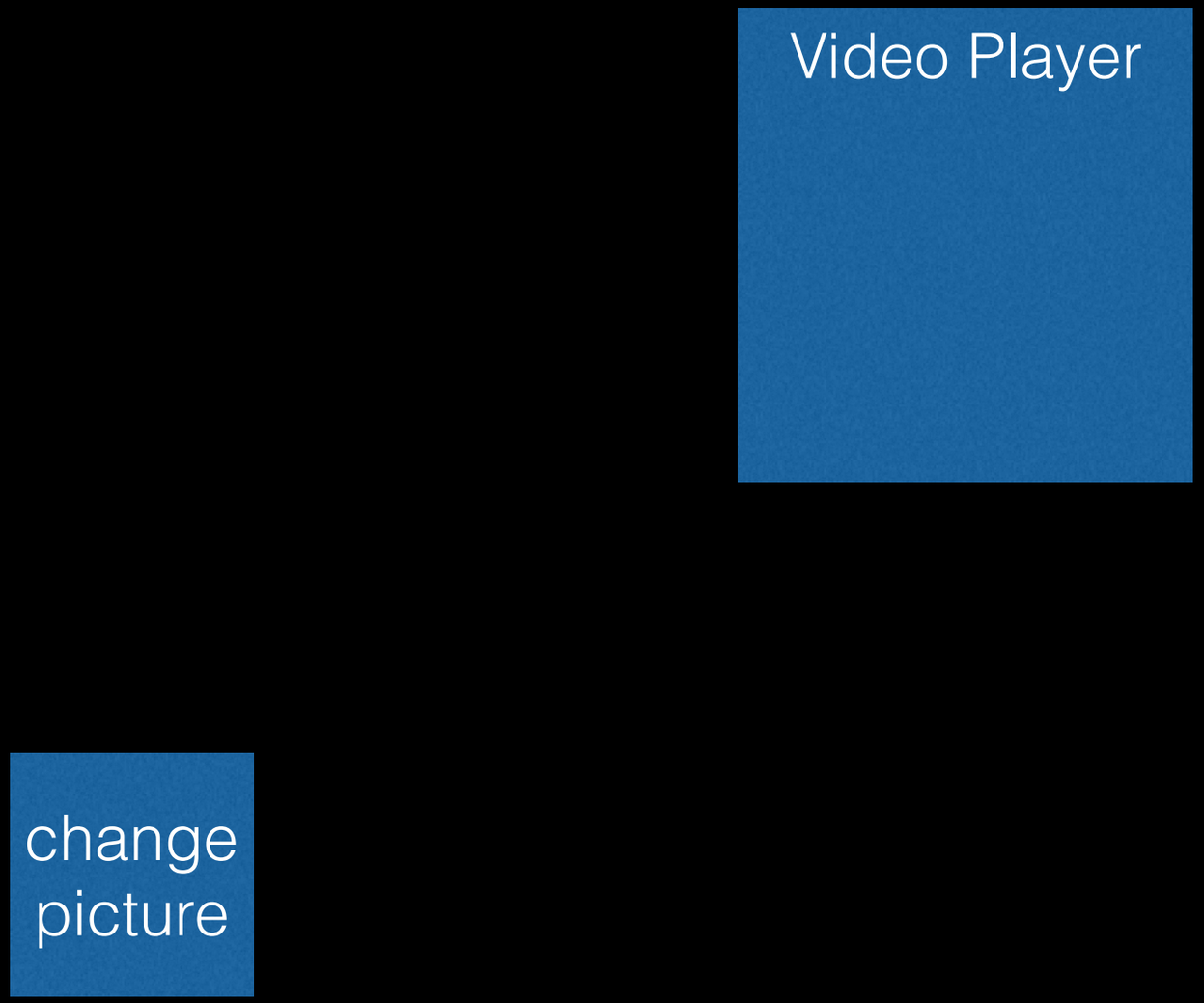
UCL

Automated Software Transplantation

Alexandru Marginean

UCL

Why Autotransplantation?



Why Autotransplantation?

change picture

Video Player

Why Autotransplantation?

Why not handle H.264?

change picture

Video Player

Why Autotransplantation?

Why not handle H.264?

change picture

Video Player

Why Autotransplantation?

Why not handle H.264?

change picture

Video Player

Why Autotransplantation?

Why not handle H.264?

change picture

Video Player

Start from scratch

Why Autotransplantation?

Why not handle H.264?

change picture

Video Player

Start from scratch

Why Autotransplantation?

Why not handle H.264?

change picture

Video Player

Check open source repositories

Start from scratch

Why Autotransplantation?

Why not handle H.264?

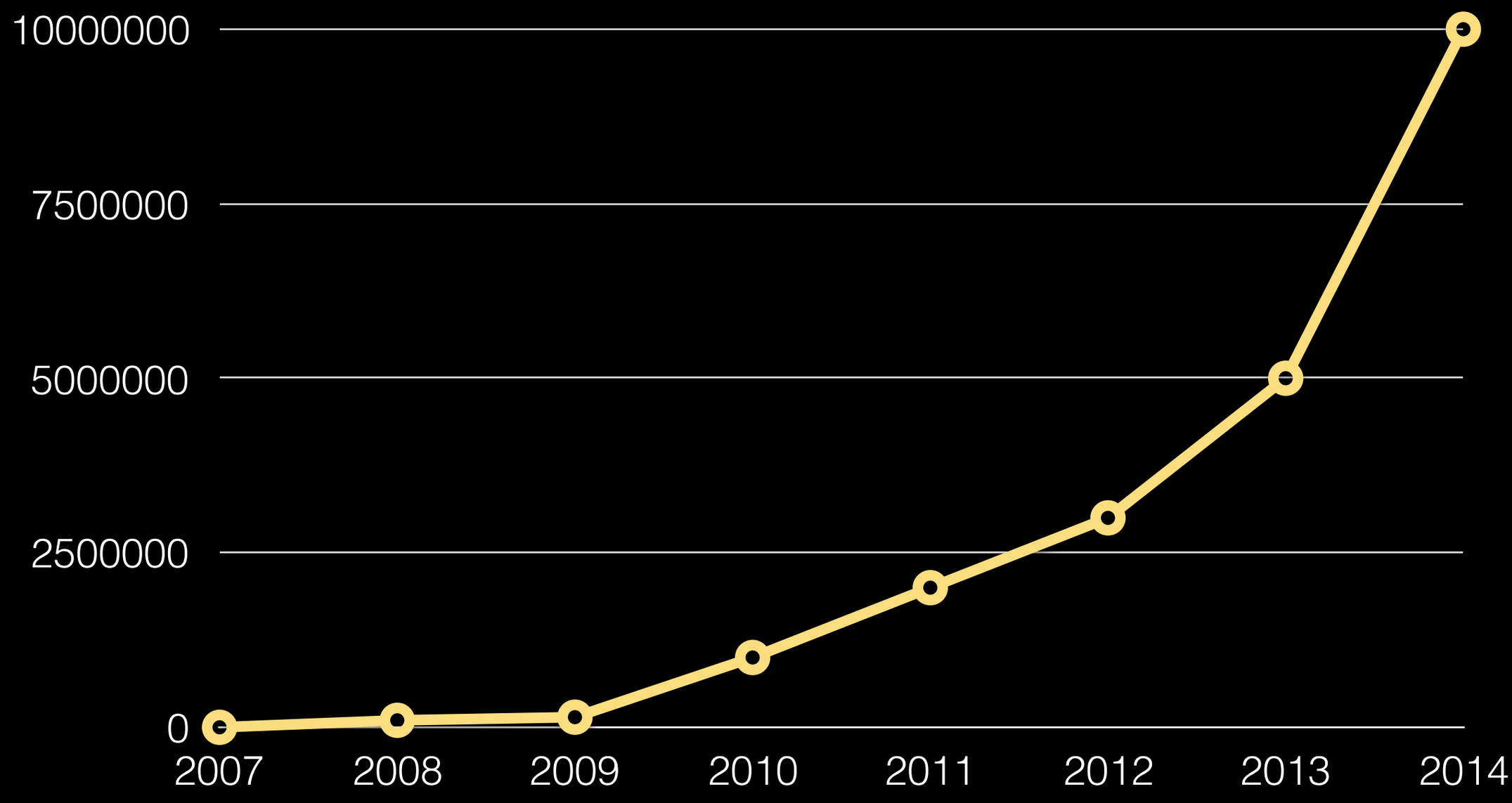
change picture

Video Player

Check open source repositories

Start from scratch

Open Source Repositories

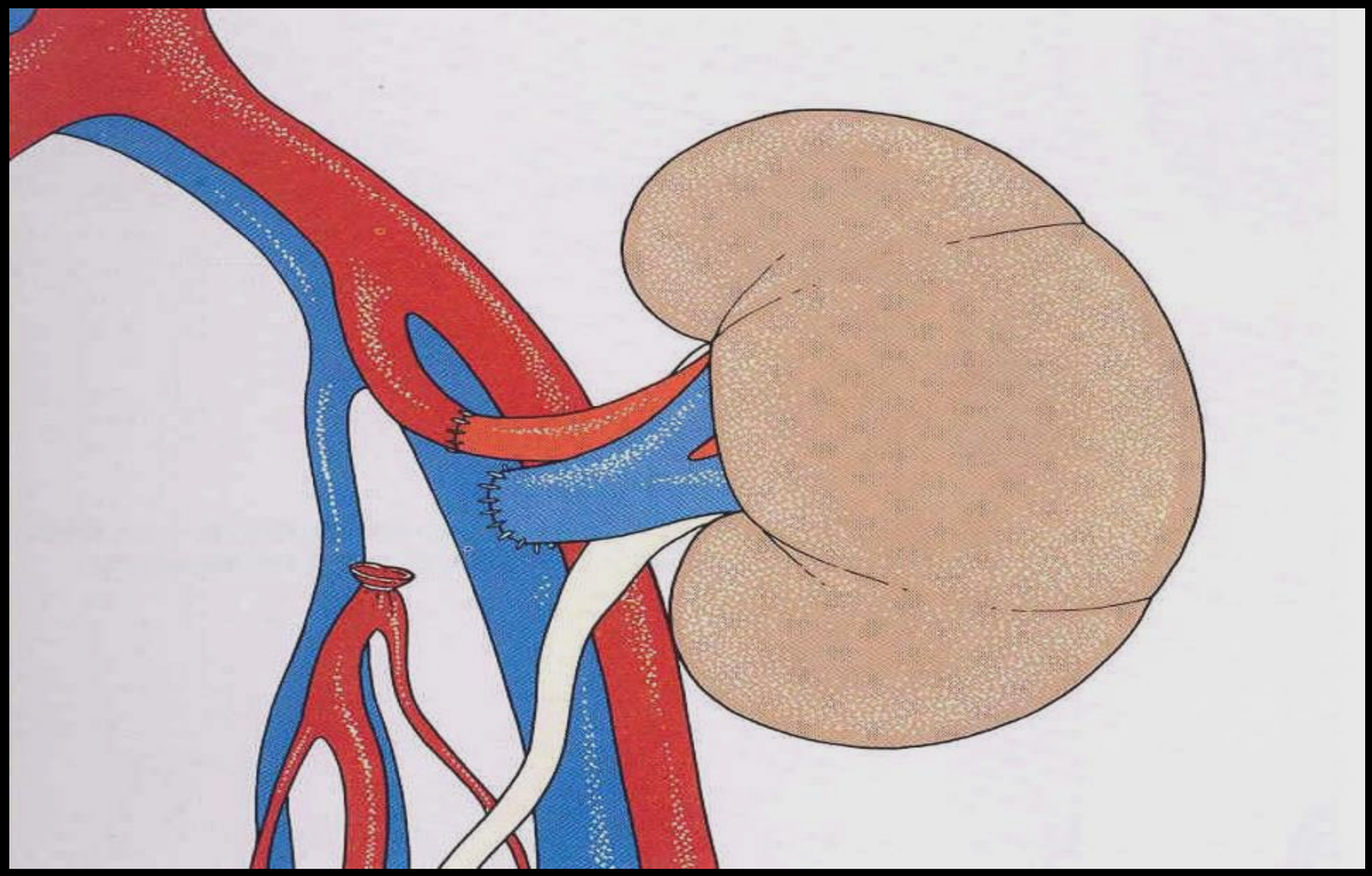


Github Popularity (repositories)

Motivation

- A lot of time is waisted in extending the functionality of an existing software system.
- Clone detection, code migration, code salvaging, reuse, dependency analyse, feature extraction techniques.
- The overall process is still manual, tedious and error prone.
- A lot of functionalities required for a new software, is already available on source code repositories.

Human Organ Transplantation



Automated Software Transplantation

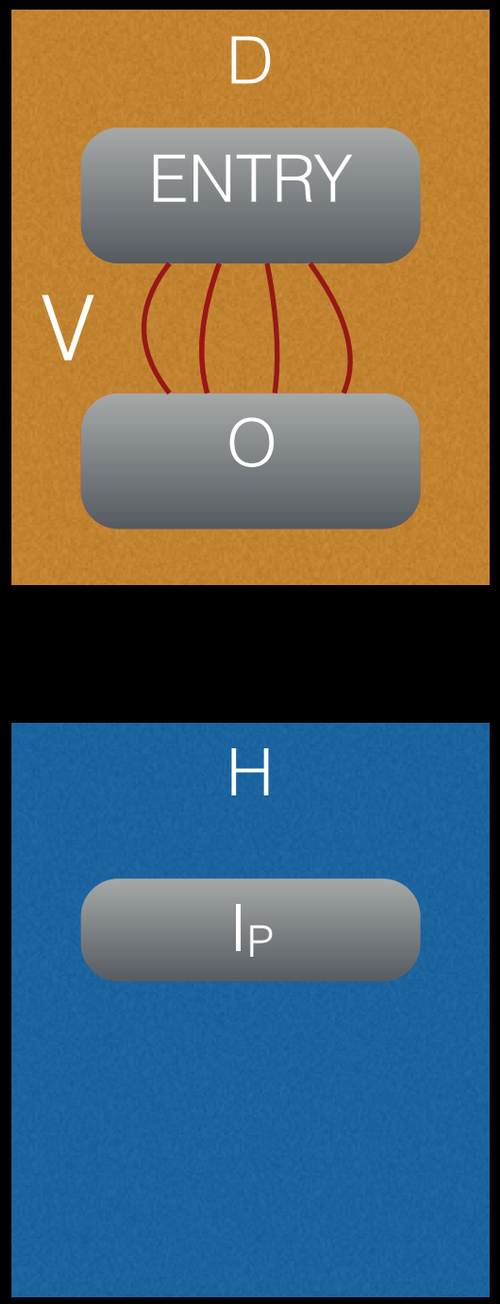
Donor

Host

Organ

Vein

I_P



Automated Software Transplantation

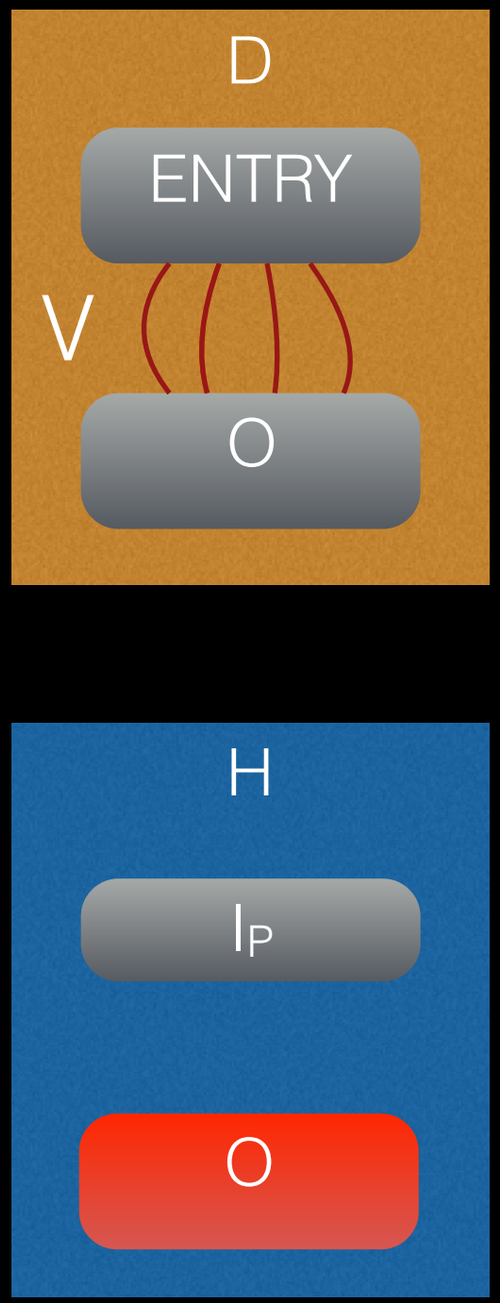
Donor

Host

Organ

Vein

I_P



Automated Software Transplantation

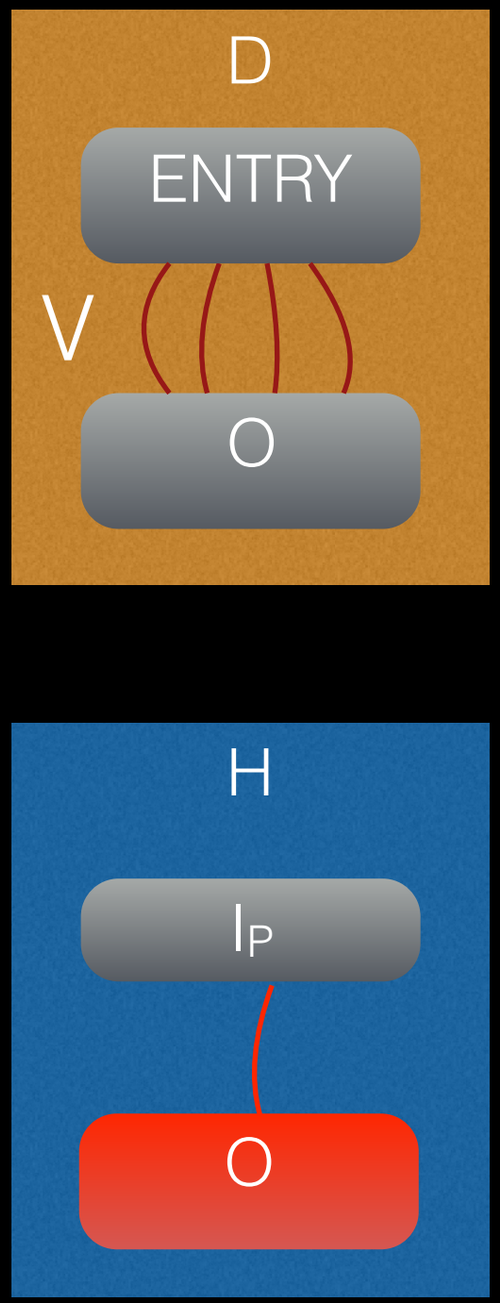
Donor

Host

Organ

Vein

I_P



Video Encoder

~100 players

H.264

...

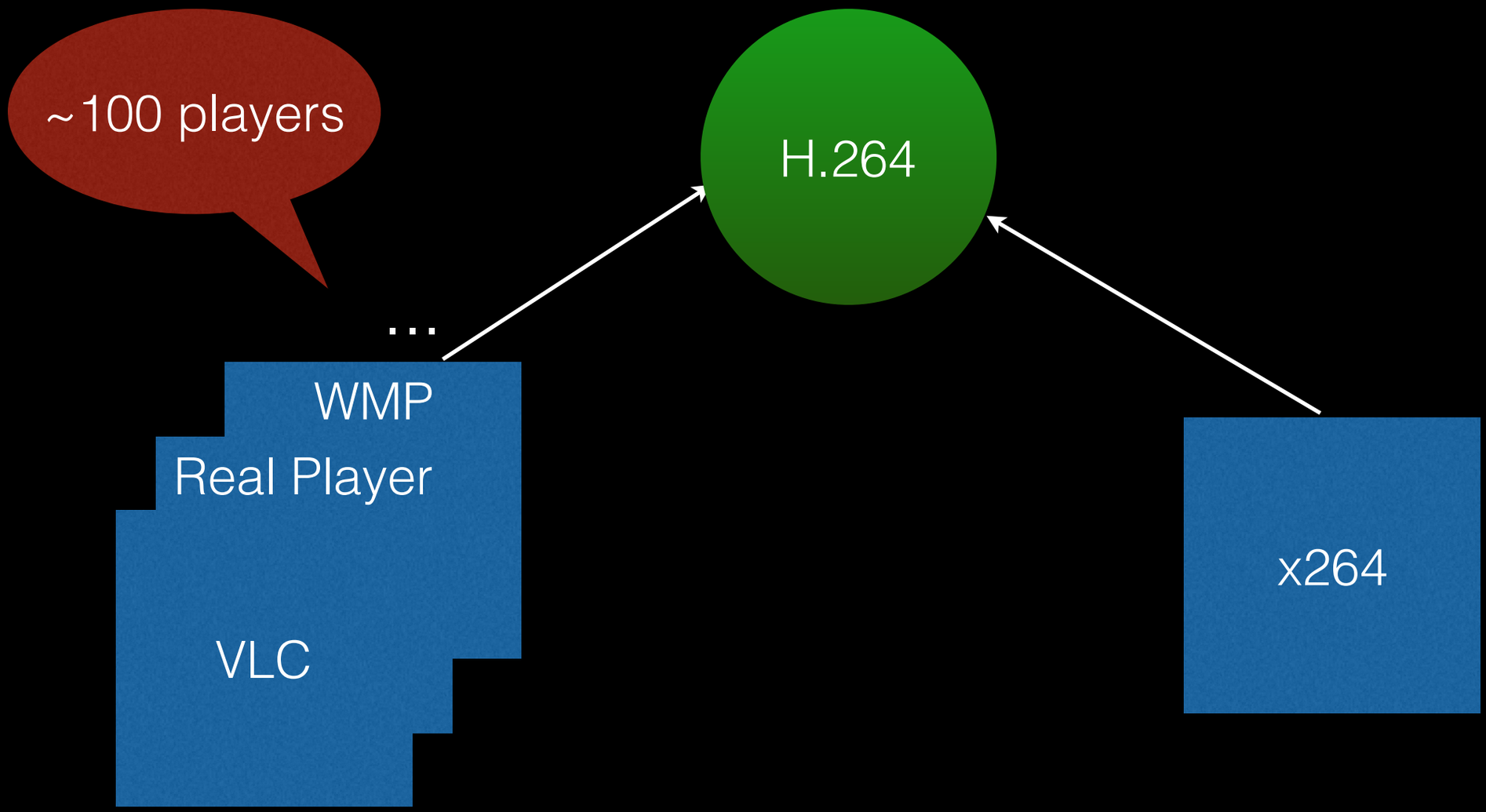
WMP

Real Player

VLC

x264

Video Encoder

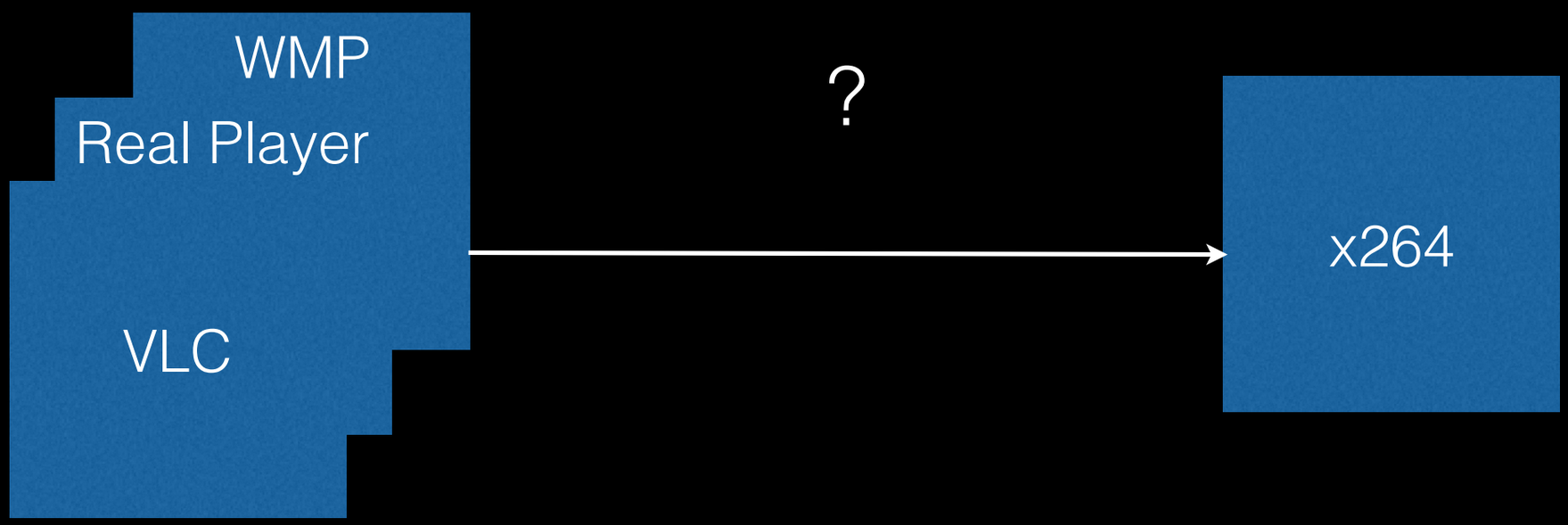


Video Encoder

~100 players

H.264

...

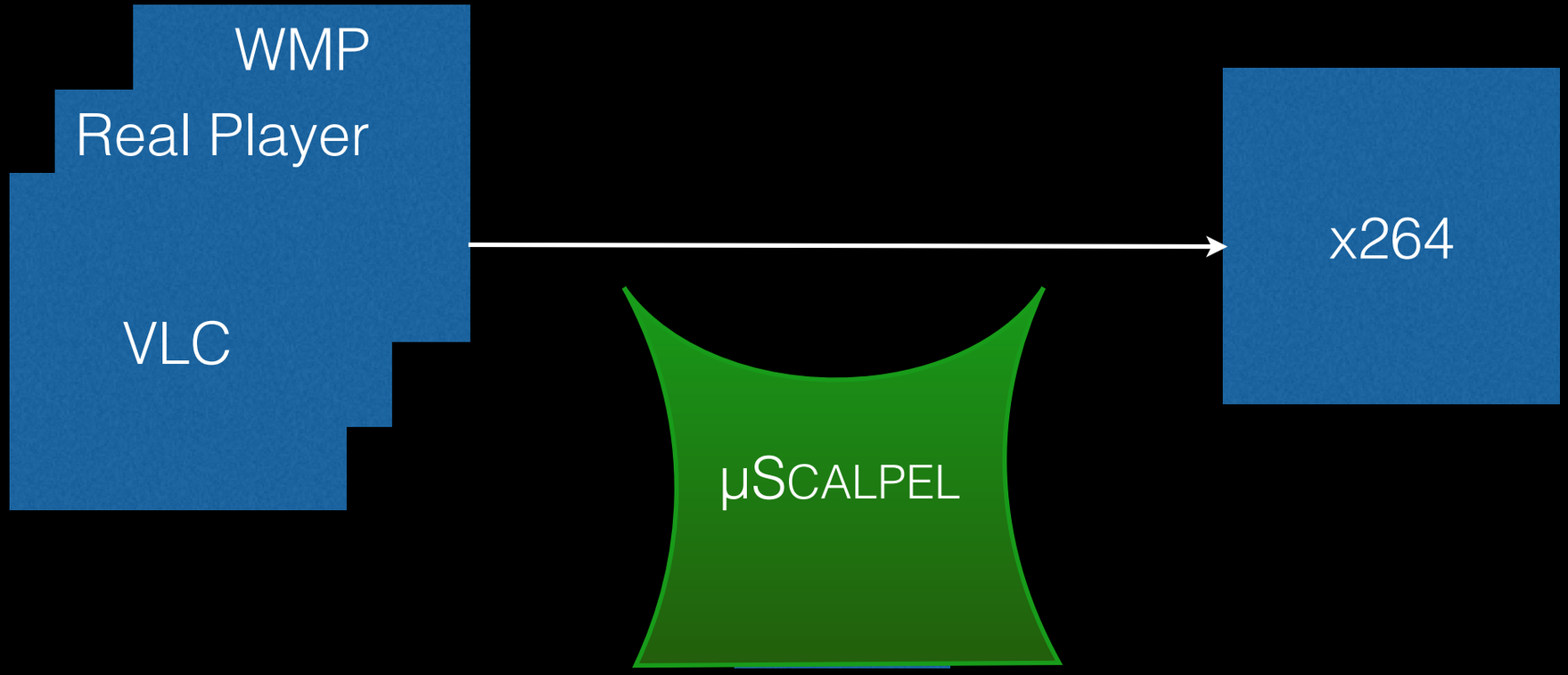


Video Encoder

~100 players

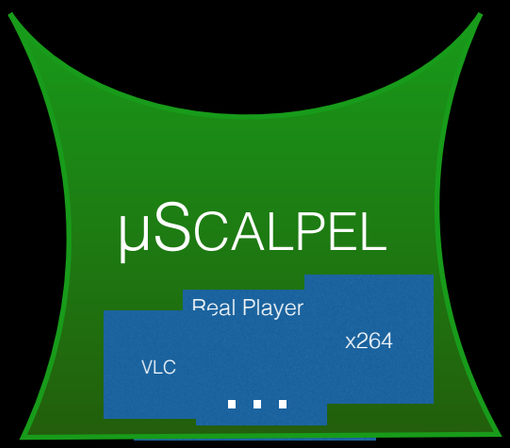
H.264

...

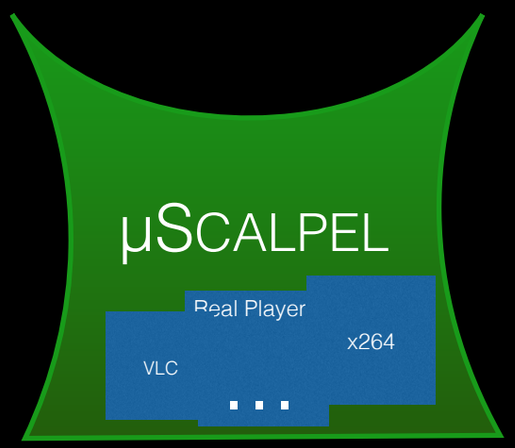


Video Encoder

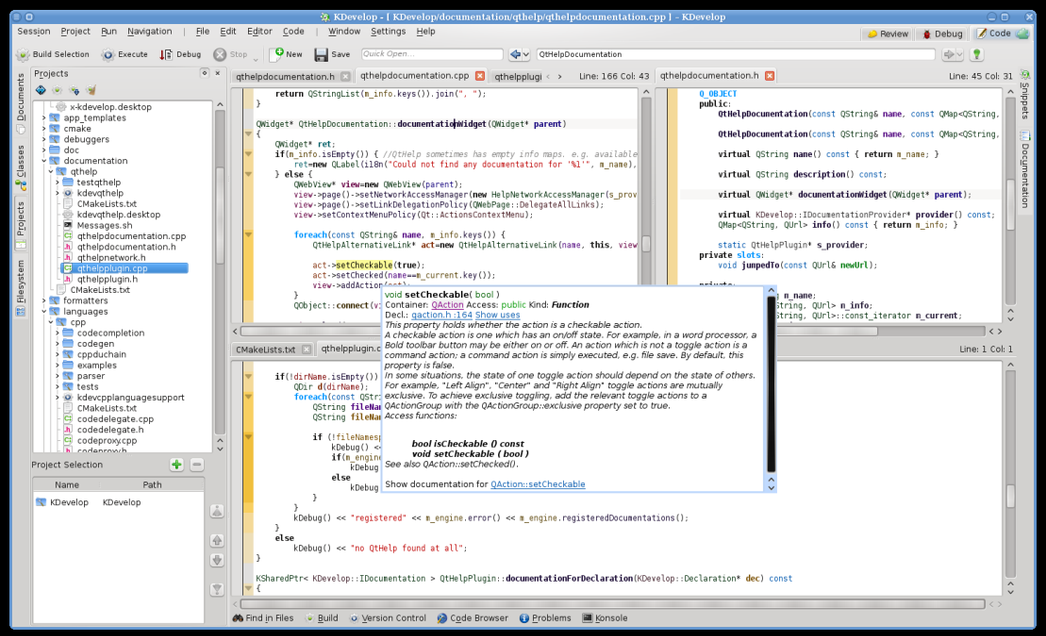
H.264



Video Encoder



Kate

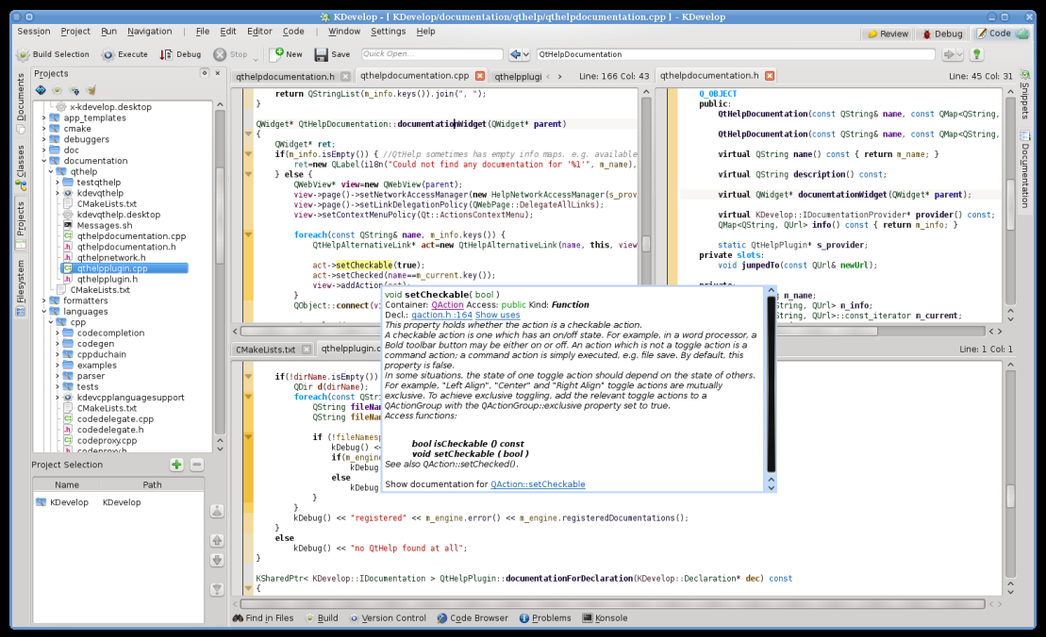


μSCALPEL

Kate

C Layout Feature?

C Call Graphs?

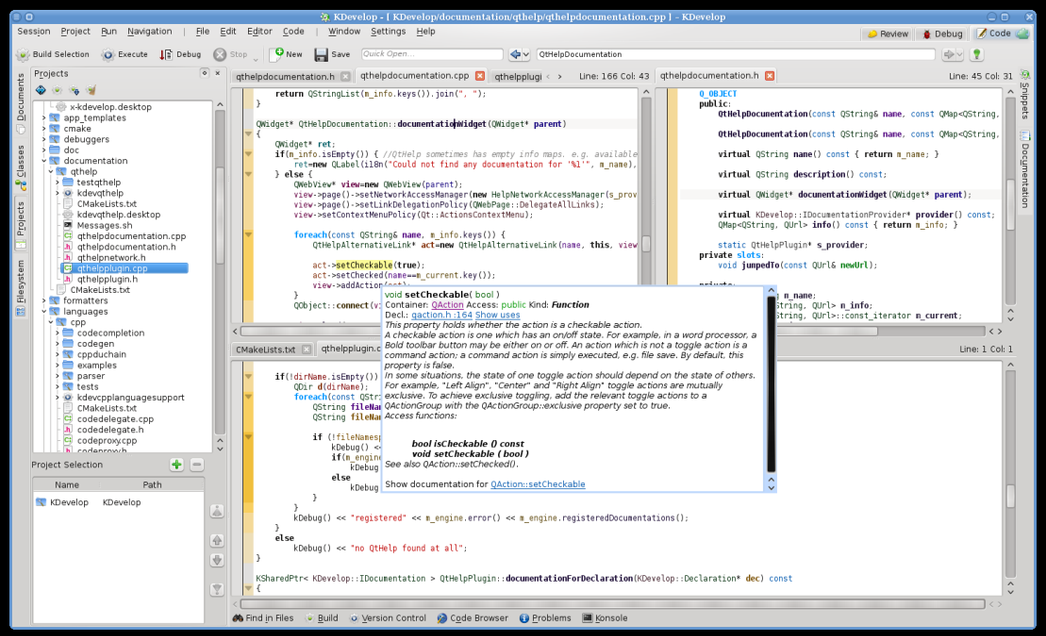


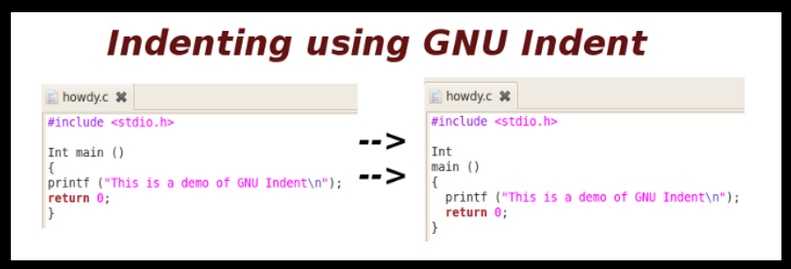
μSCALPEL

Kate

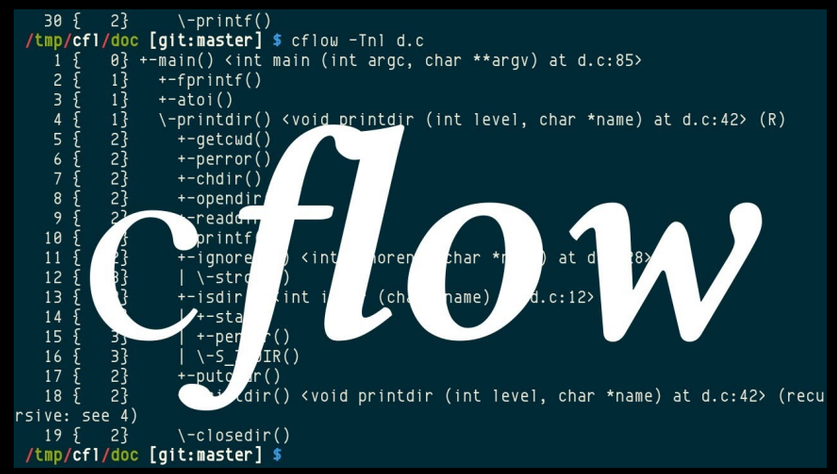
C Layout Feature?

C Call Graphs?





μSCALPEL

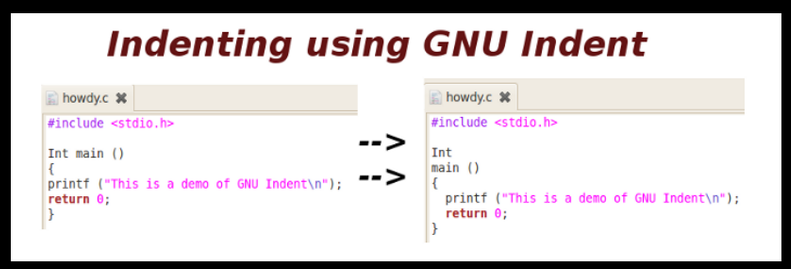


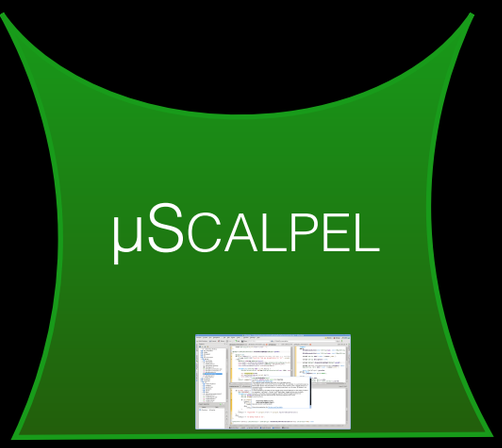
cflow

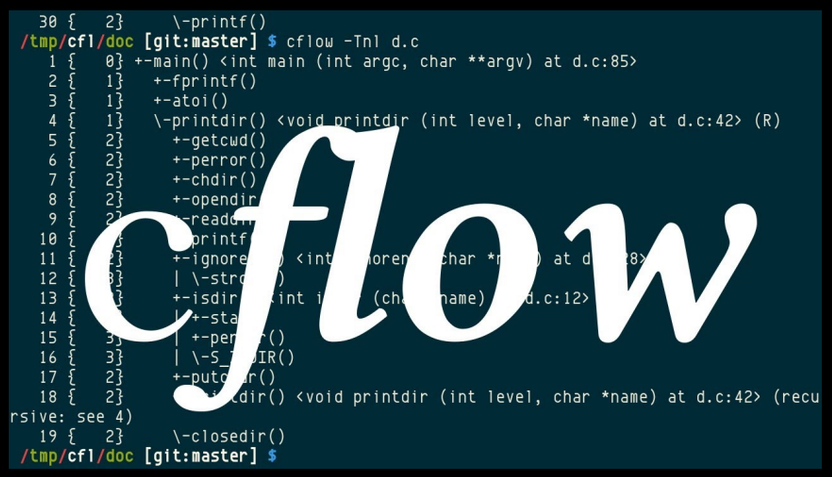
UCL

Alexandru Marginean - Automated Software Transplantation

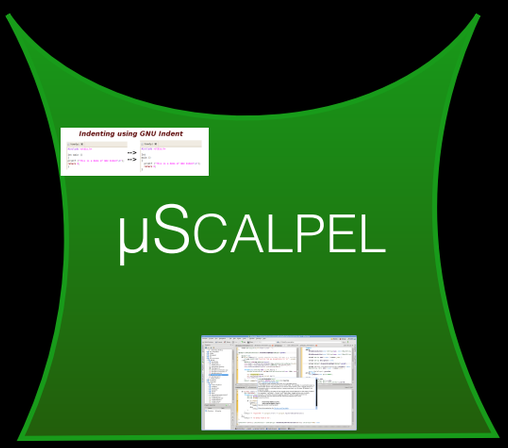
Kate

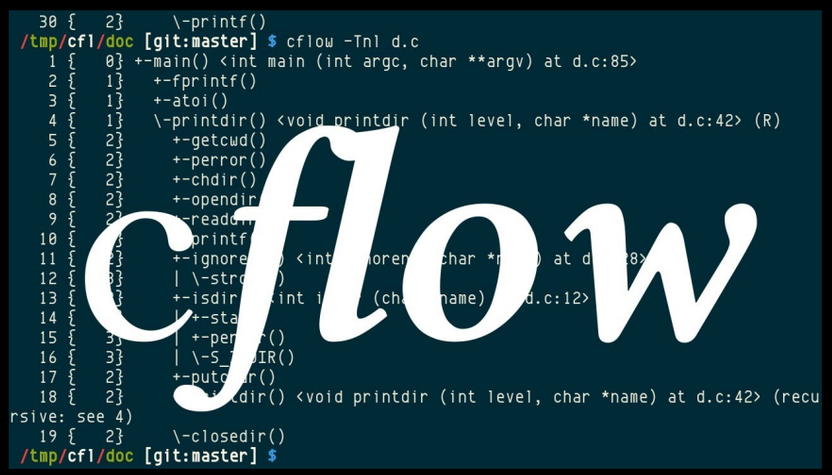

Indenting using GNU Indent
howdy.c
#include <stdio.h>
Int main ()
{
printf ("This is a demo of GNU Indent\n");
return 0;
}
-->
-->
howdy.c
#include <stdio.h>
Int
main ()
{
printf ("This is a demo of GNU Indent\n");
return 0;
}


µSCALPEL

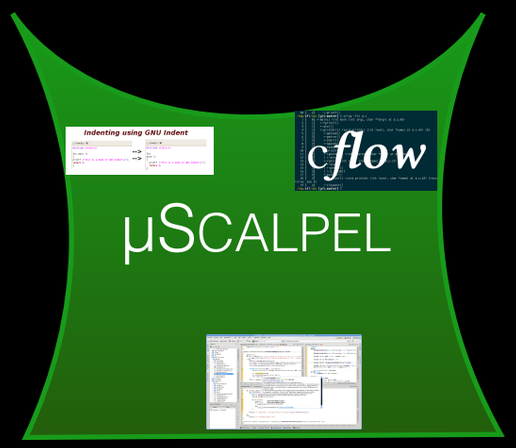

30 { 2} \-printf()
/tmp/cfl/doc [git:master] $ cflow -Tnl d.c
1 { 0} +-main() <int main (int argc, char **argv) at d.c:85>
2 { 1} +-fprintf()
3 { 1} +-atoi()
4 { 1} \-printdir() <void printdir (int level, char *name) at d.c:42> (R)
5 { 2} +-getcwd()
6 { 2} +-perror()
7 { 2} +-chdir()
8 { 2} +-opendir()
9 { 2} +-readdir()
10 { 2} +-printf()
11 { 2} +-ignore() <int ignore(char *name) at d.c:28>
12 { 3} | \-strcmp()
13 { 2} +-isdir() <int isdir (char *name) at d.c:12>
14 { 3} | +-stat()
15 { 3} | +-perror()
16 { 3} | \-S_ISDIR()
17 { 2} +-putchar()
18 { 2} +-closedir() <void printdir (int level, char *name) at d.c:42> (recursive: see 4)
19 { 2} \-closedir()
/tmp/cfl/doc [git:master] $
cflow

Kate

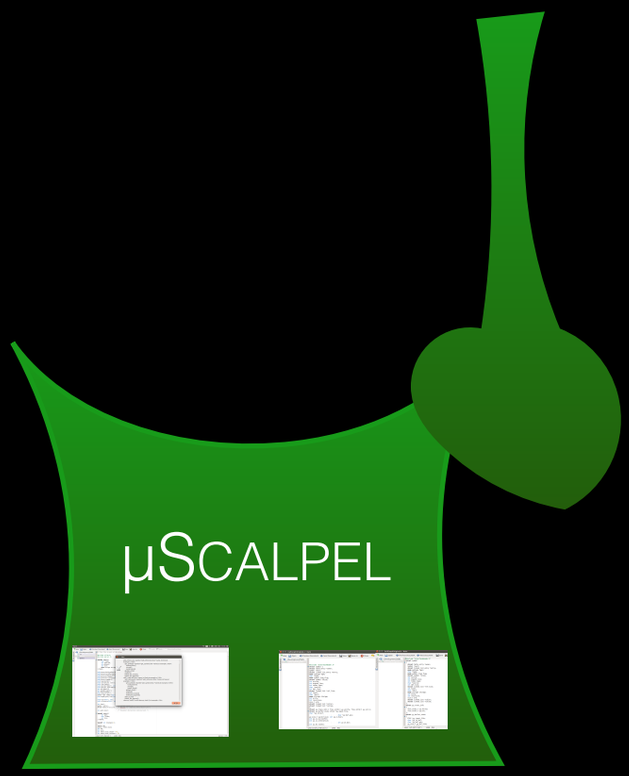

Indenting using GNU Indent
μSCALPEL


30 { 2}   \-printf()
/tmp/cfl/doc [git:master] $ cflow -Tnl d.c
1 { 0} +-main() <int main (int argc, char **argv) at d.c:85>
2 { 1}   +-fprintf()
3 { 1}   +-atoi()
4 { 1}   \-printdir() <void printdir (int level, char *name) at d.c:42> (R)
5 { 2}   +-getcwd()
6 { 2}   +-perror()
7 { 2}   +-chdir()
8 { 2}   +-opendir()
cflow
/tmp/cfl/doc [git:master] $

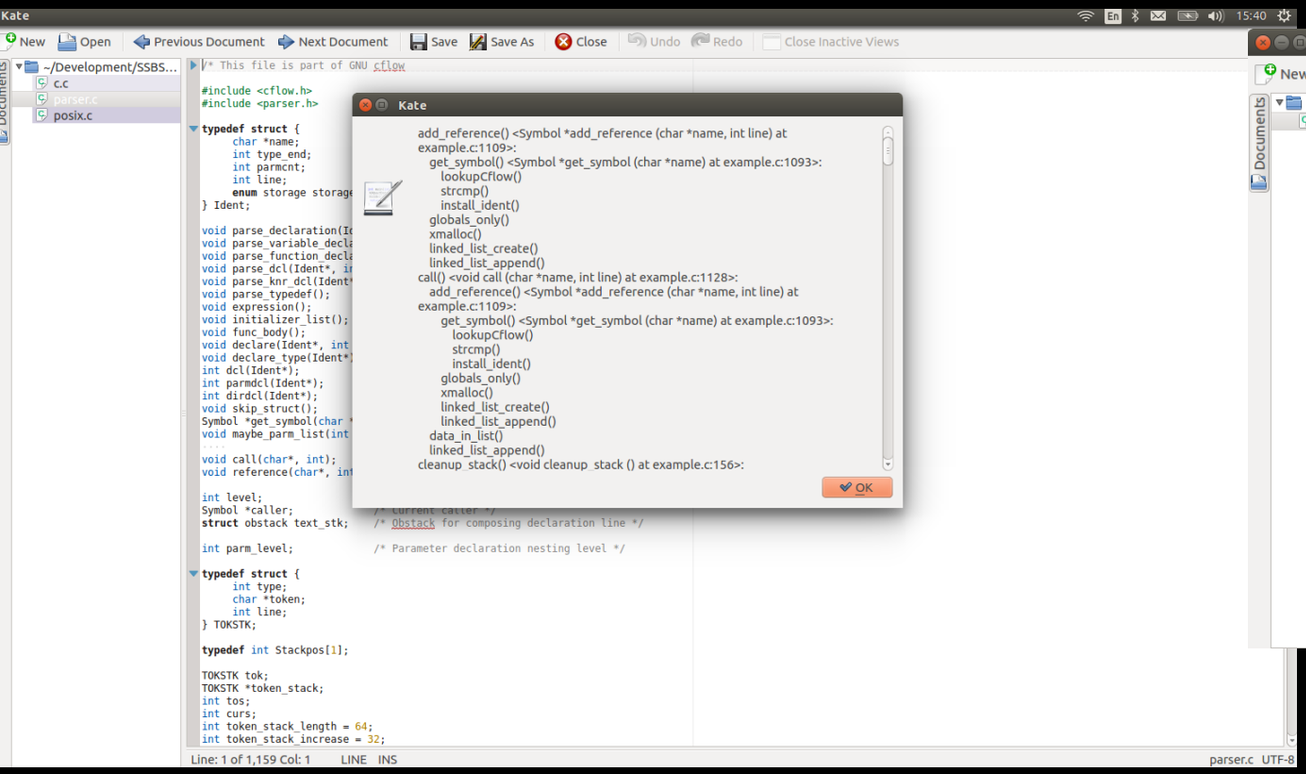
Kate

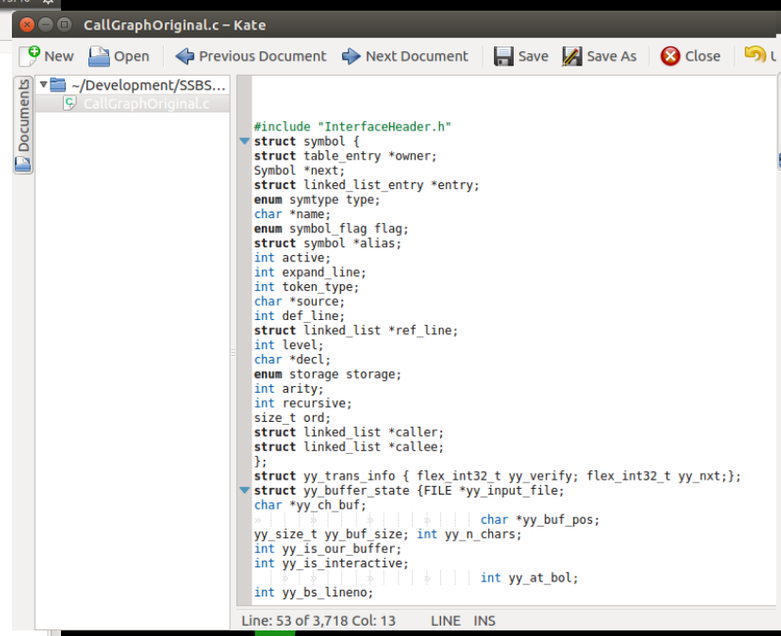


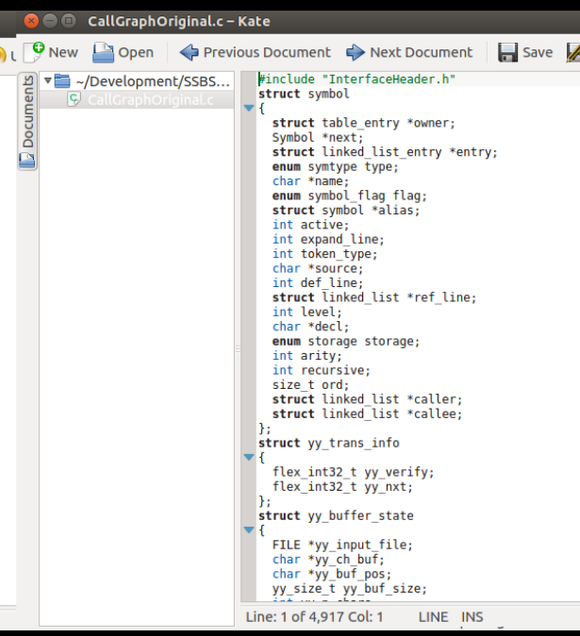
Kate

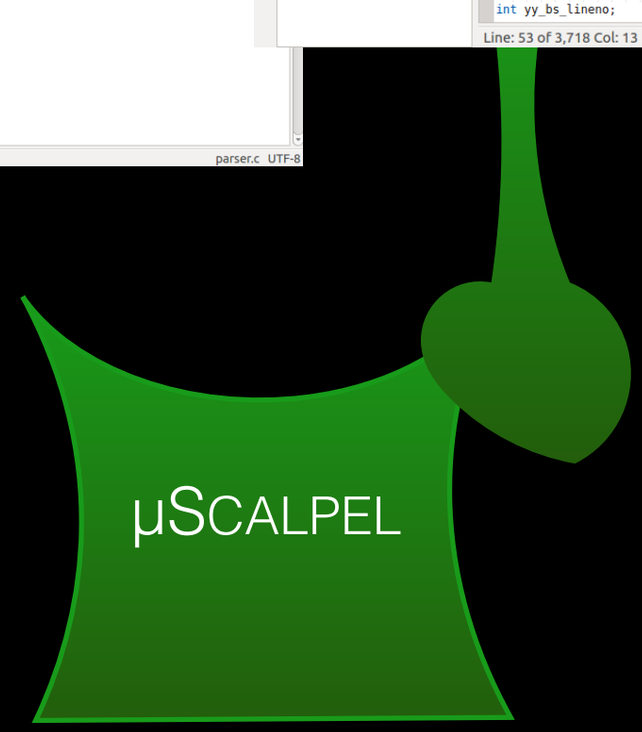


Kate

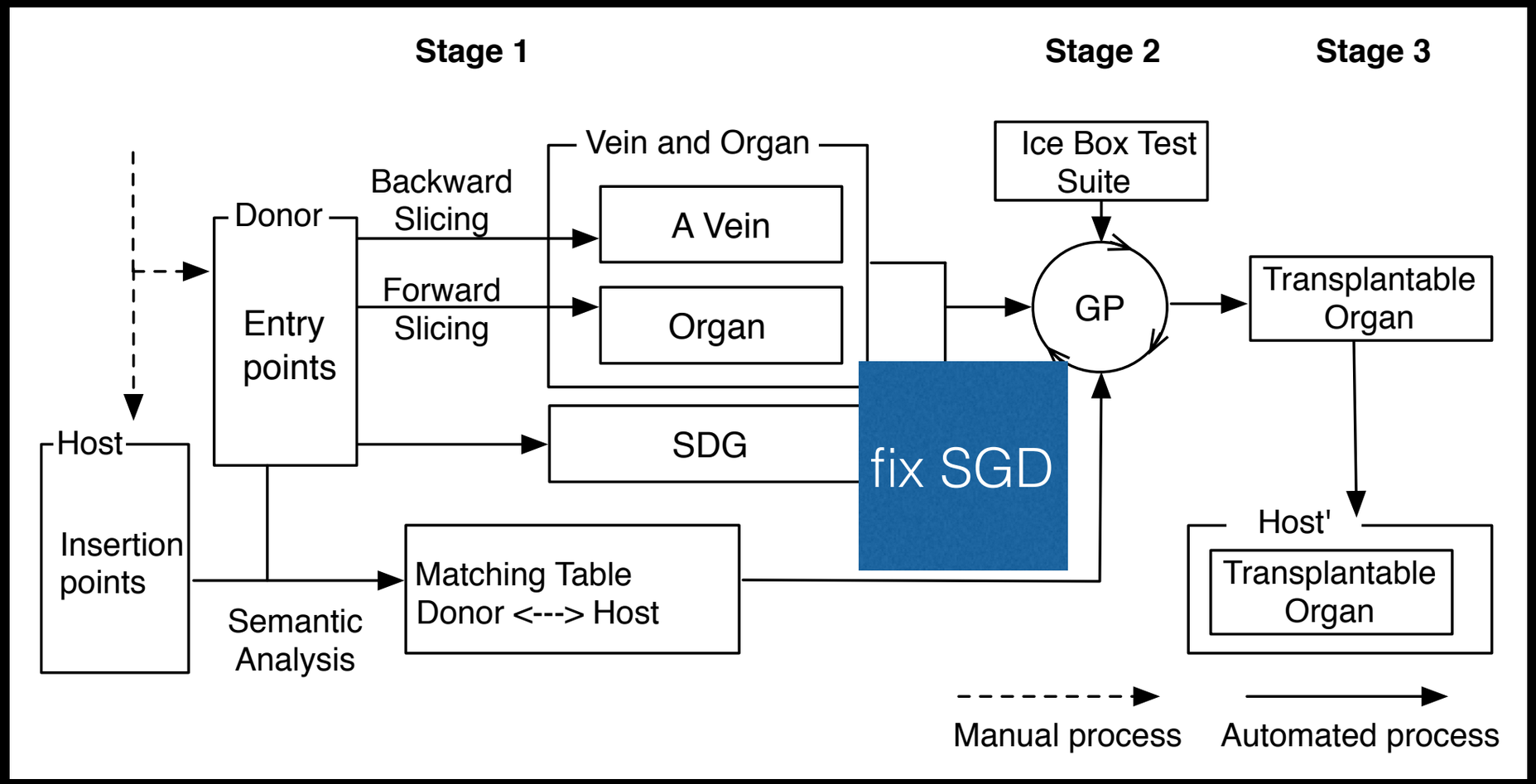

Kate
New
Open
Previous Document
Next Document
Save
Save As
Close
Undo
Redo
Close Inactive Views
~/Development/SSBS...
c.c
parser.c
posix.c
/* This file is part of GNU cflow
#include <cflow.h>
#include <parser.h>
typedef struct {
char *name;
int type_end;
int parmcnt;
int line;
enum storage storage;
} Ident;
void parse_declaration(Id
void parse_variable_decla
void parse_function_decla
void parse_dcl(Ident*, in
void parse_knr_dcl(Ident*
void parse_typedef();
void expression();
void initializer_list();
void func_body();
void declare(Ident*, int
void declare_type(Ident*)
int dcl(Ident*);
int parmdcl(Ident*);
int dirdcl(Ident*);
void skip_struct();
Symbol *get_symbol(char *
void maybe_parm_list(int
void call(char*, int);
void reference(char*, int
int level;
Symbol *caller;
struct obstack text_stk; /* Obstack for composing declaration line */
int parm_level; /* Parameter declaration nesting level */
typedef struct {
int type;
char *token;
int line;
} TOKSTK;
typedef int Stackpos[1];
TOKSTK tok;
TOKSTK *token_stack;
int tos;
int curs;
int token_stack_length = 64;
int token_stack_increase = 32;
Line: 1 of 1,159 Col: 1 LINE INS
parser.c UTF-8
Kate
add_reference() <Symbol *add_reference (char *name, int line) at example.c:1109>:
get_symbol() <Symbol *get_symbol (char *name) at example.c:1093>:
lookupCflow()
strcmp()
install_ident()
globals_only()
xmalloc()
linked_list_create()
linked_list_append()
call() <void call (char *name, int line) at example.c:1128>:
add_reference() <Symbol *add_reference (char *name, int line) at example.c:1109>:
get_symbol() <Symbol *get_symbol (char *name) at example.c:1093>:
lookupCflow()
strcmp()
install_ident()
globals_only()
xmalloc()
linked_list_create()
linked_list_append()
data_in_list()
linked_list_append()
cleanup_stack() <void cleanup_stack () at example.c:156>:
OK


CallGraphOriginal.c – Kate
New
Open
Previous Document
Next Document
Save
Save As
Close
~/Development/SSBS...
CallGraphOriginal.c
#include "InterfaceHeader.h"
struct symbol {
struct table_entry *owner;
Symbol *next;
struct linked_list_entry *entry;
enum symtype type;
char *name;
enum symbol_flag flag;
struct symbol *alias;
int active;
int expand_line;
int token_type;
char *source;
int def_line;
struct linked_list *ref_line;
int level;
char *decl;
enum storage storage;
int arity;
int recursive;
size_t ord;
struct linked_list *caller;
struct linked_list *callee;
};
struct yy_trans_info { flex_int32_t yy_verify; flex_int32_t yy_nxt;};
struct yy_buffer_state {FILE *yy_input_file;
char *yy_ch_buf;
char *yy_buf_pos;
yy_size_t yy_buf_size; int yy_n_chars;
int yy_is_our_buffer;
int yy_is_interactive;
int yy_at_bol;
int yy_bs_lineno;
Line: 53 of 3,718 Col: 13 LINE INS

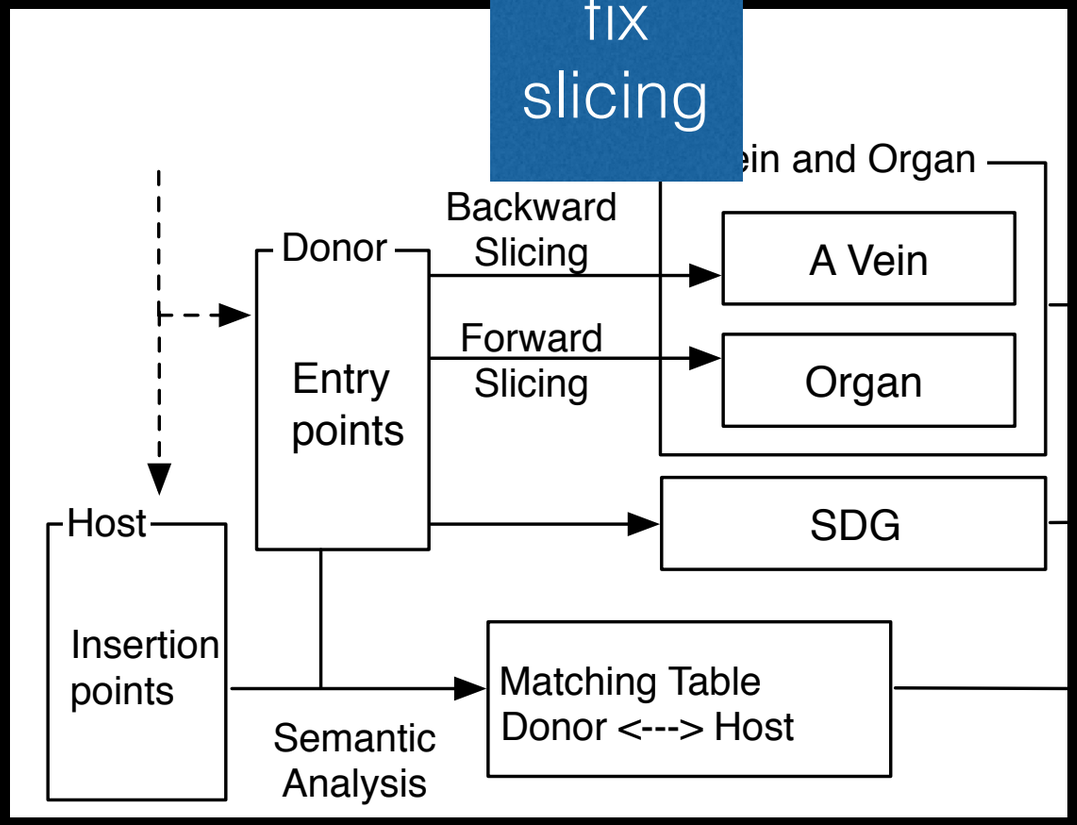

CallGraphOriginal.c – Kate
New
Open
Previous Document
Next Document
Save
~/Development/SSBS...
CallGraphOriginal.c
#include "InterfaceHeader.h"
struct symbol
{
struct table_entry *owner;
Symbol *next;
struct linked_list_entry *entry;
enum symtype type;
char *name;
enum symbol_flag flag;
struct symbol *alias;
int active;
int expand_line;
int token_type;
char *source;
int def_line;
struct linked_list *ref_line;
int level;
char *decl;
enum storage storage;
int arity;
int recursive;
size_t ord;
struct linked_list *caller;
struct linked_list *callee;
};
struct yy_trans_info
{
flex_int32_t yy_verify;
flex_int32_t yy_nxt;
};
struct yy_buffer_state
{
FILE *yy_input_file;
char *yy_ch_buf;
char *yy_buf_pos;
yy_size_t yy_buf_size;
Line: 1 of 4,917 Col: 1 LINE INS

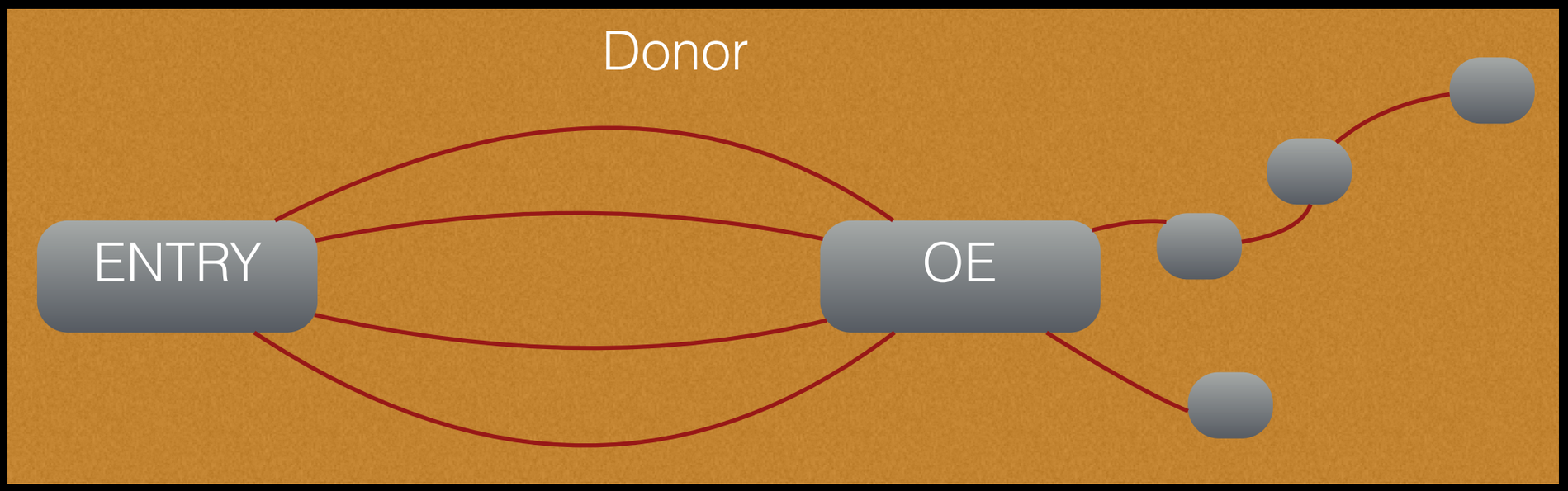

μSCALPEL

Approach

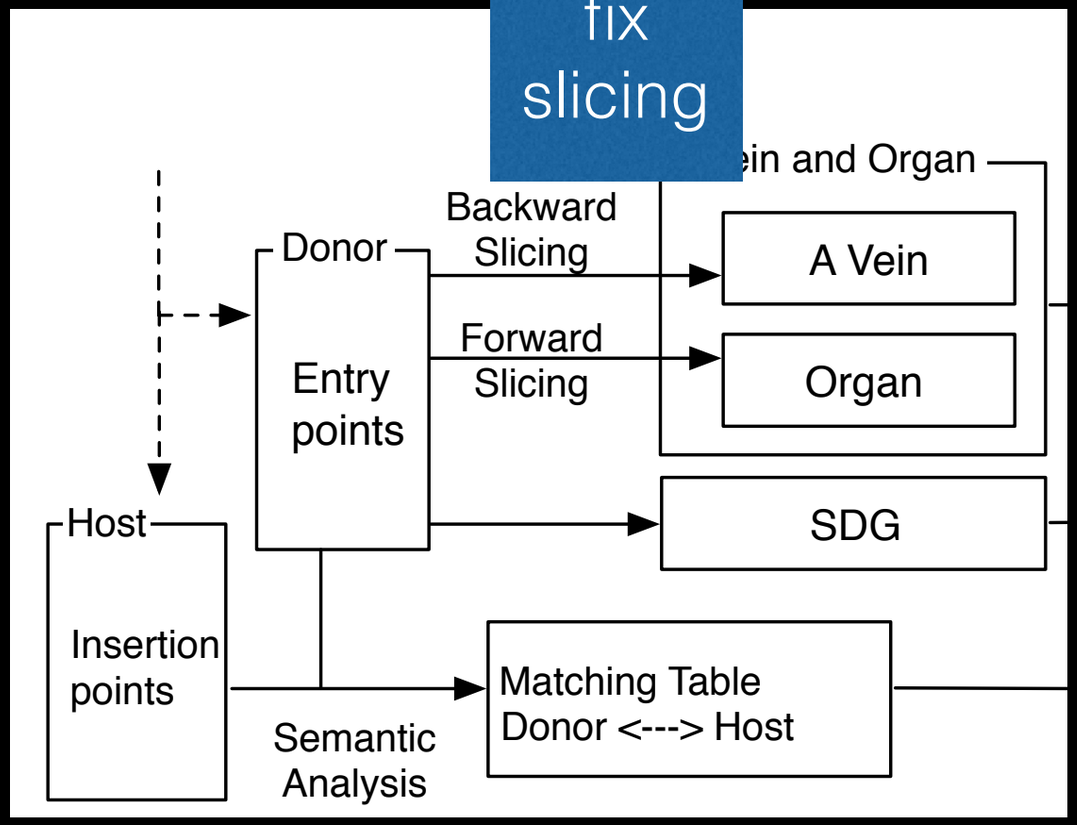

Stage 1
Stage 2
Stage 3
Donor
Entry points
Backward Slicing
Forward Slicing
Vein and Organ
A Vein
Organ
SDG
Host
Insertion points
Semantic Analysis
Matching Table
Donor <---> Host
Ice Box Test Suite
GP
fix SGD
Transplantable Organ
Host'
Transplantable Organ
Manual process
Automated process

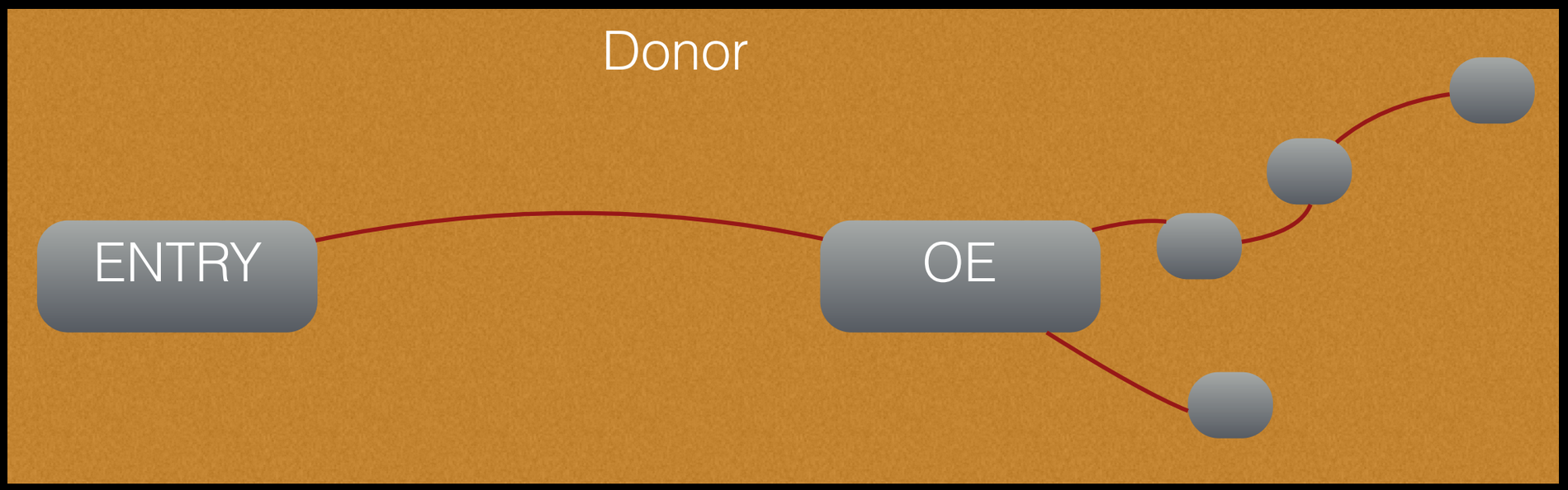
Stage 1 - Static Analysis



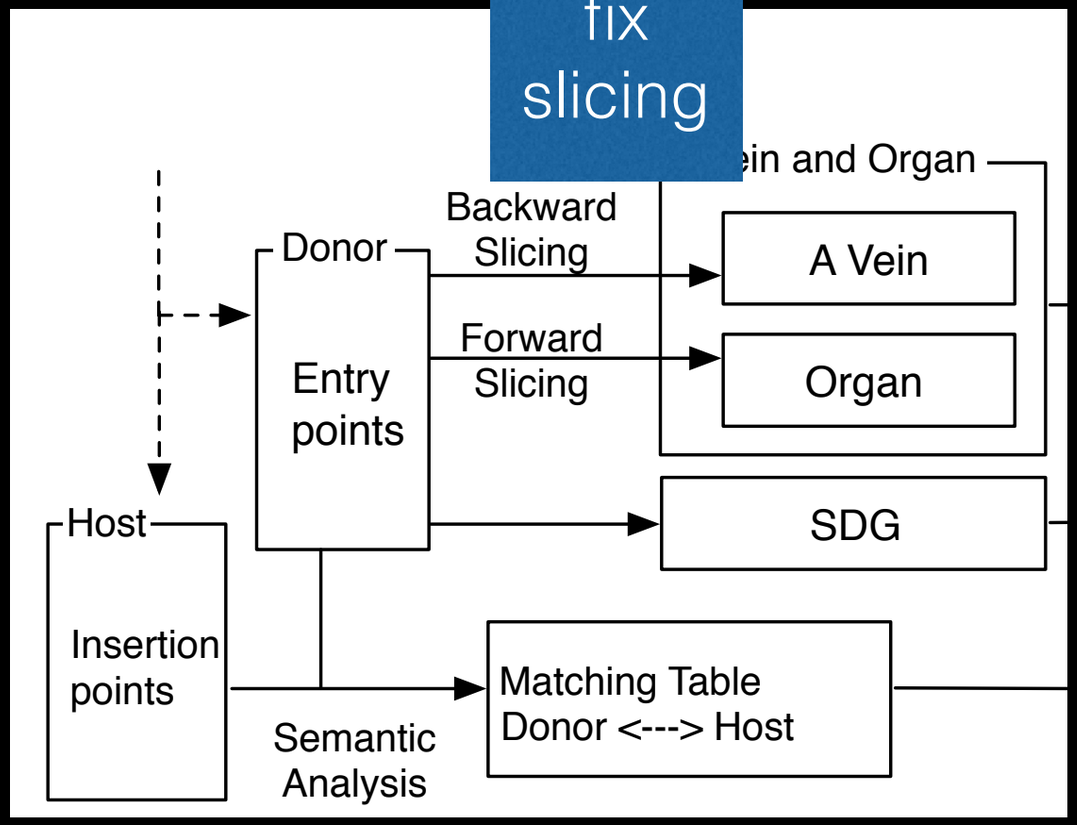


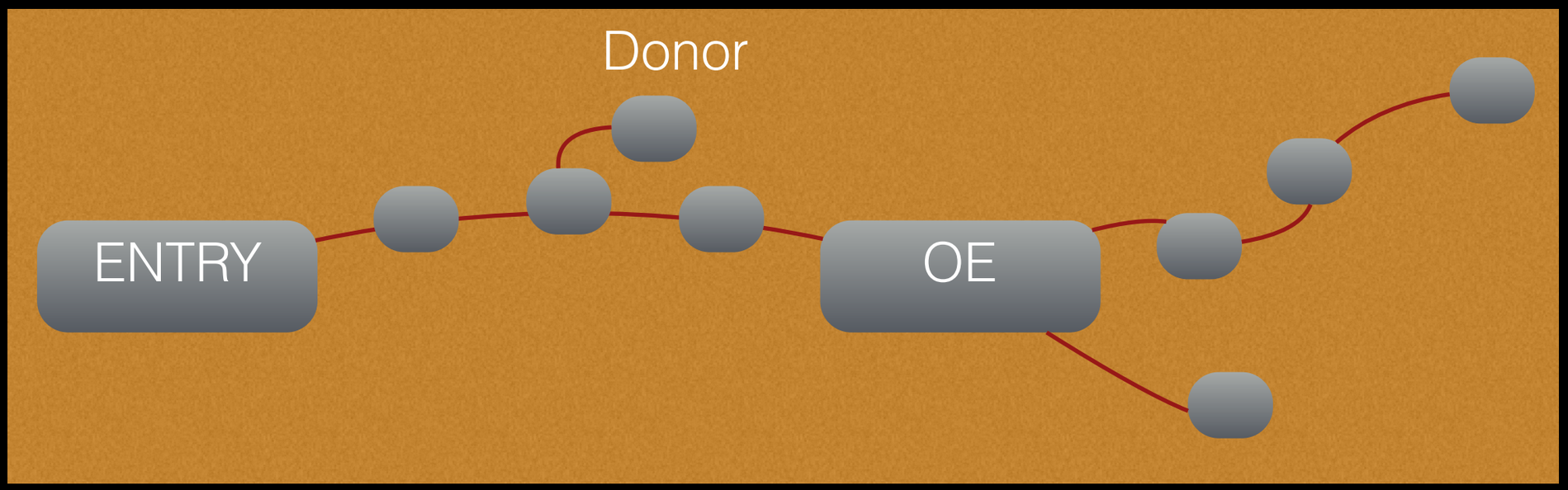
Stage 1 - Static Analysis



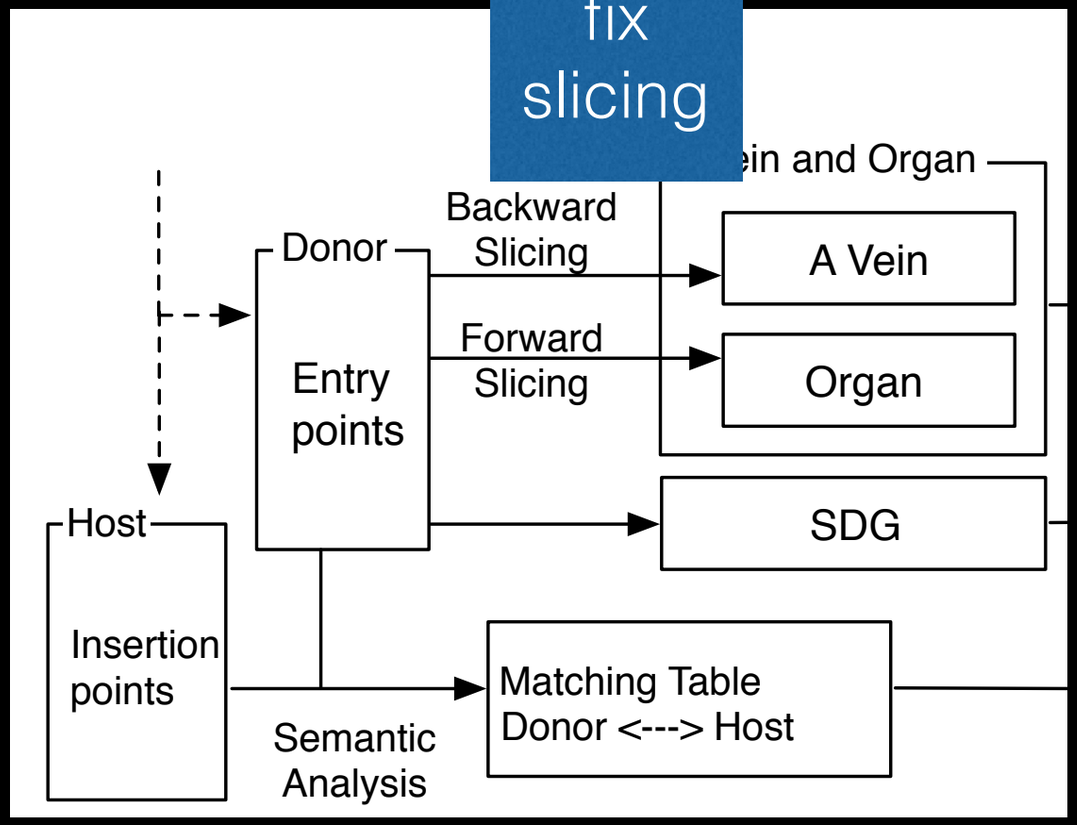


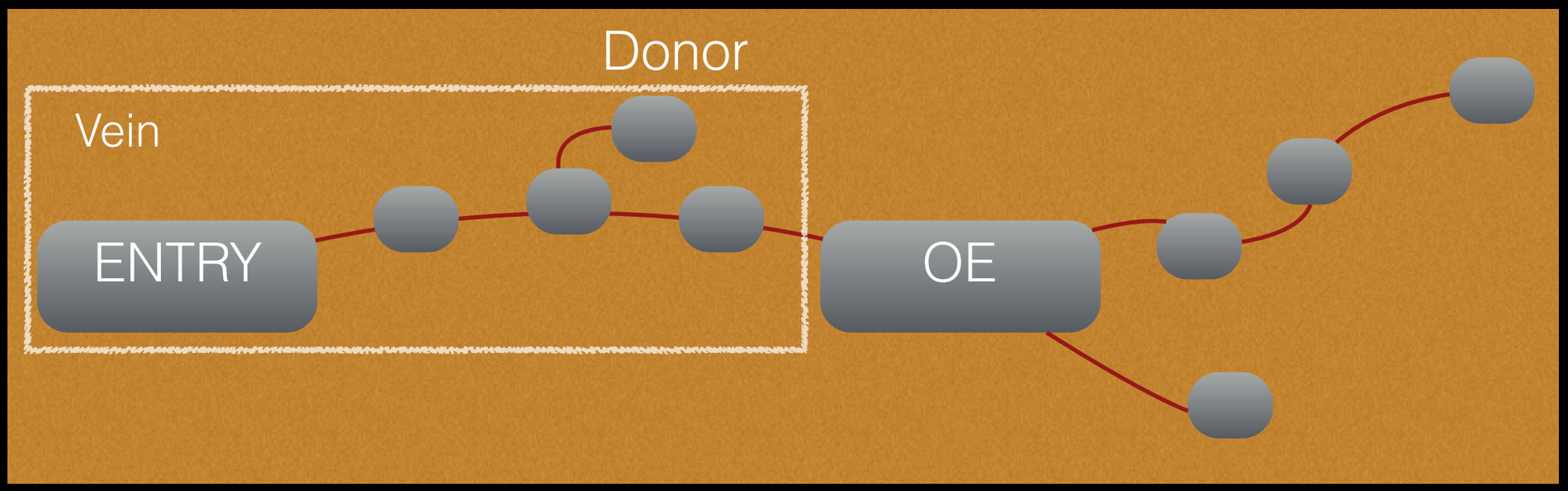
Stage 1 - Static Analysis



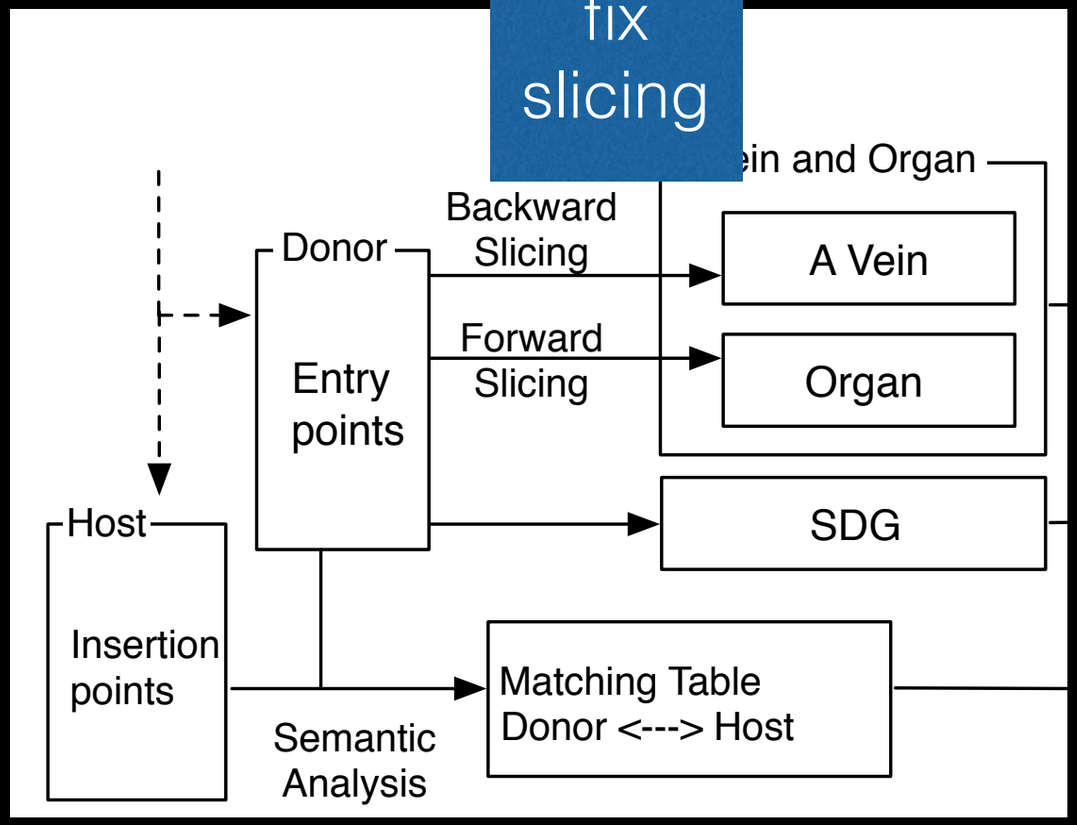


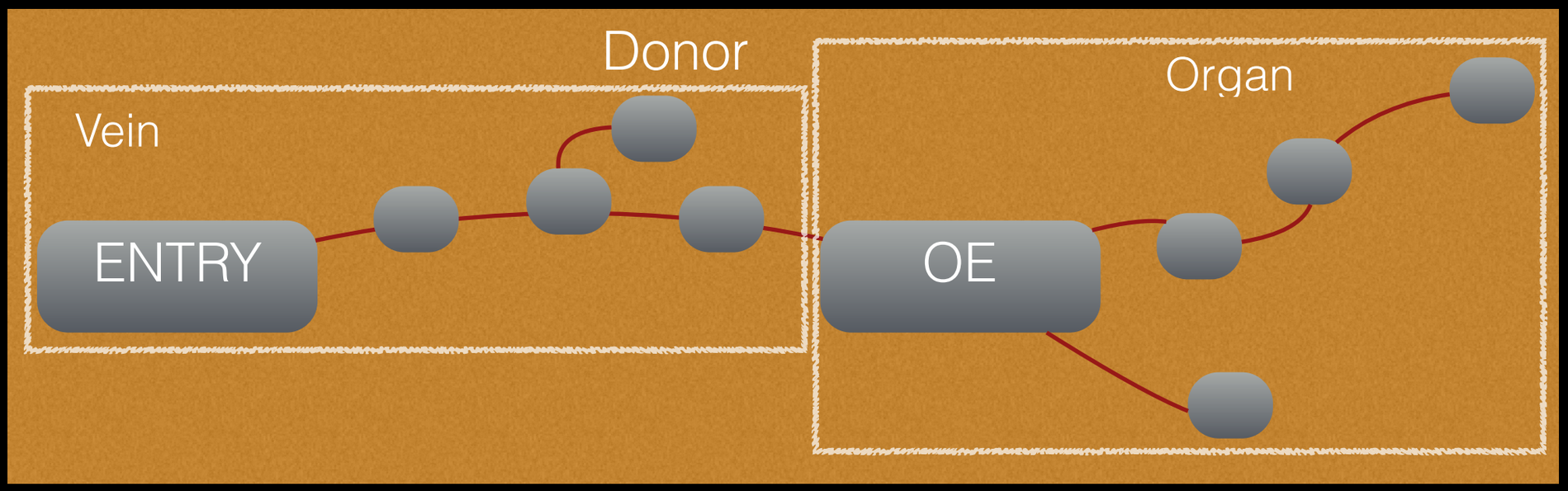
Stage 1 - Static Analysis



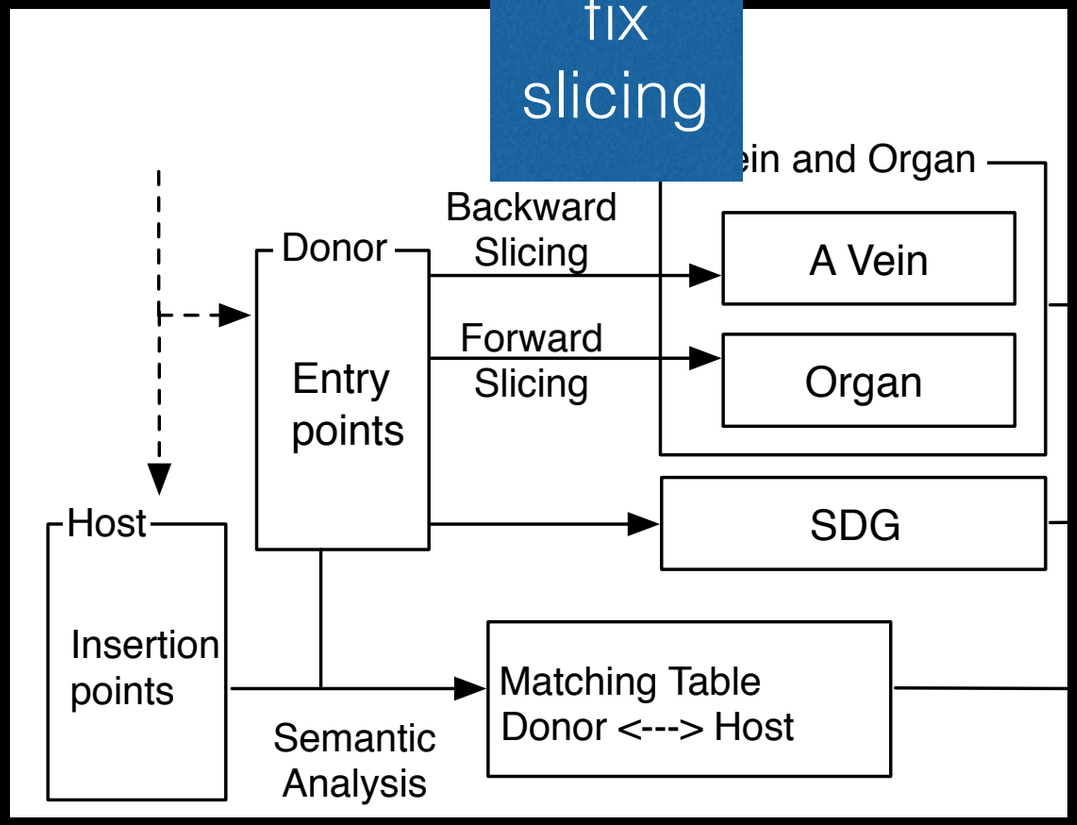


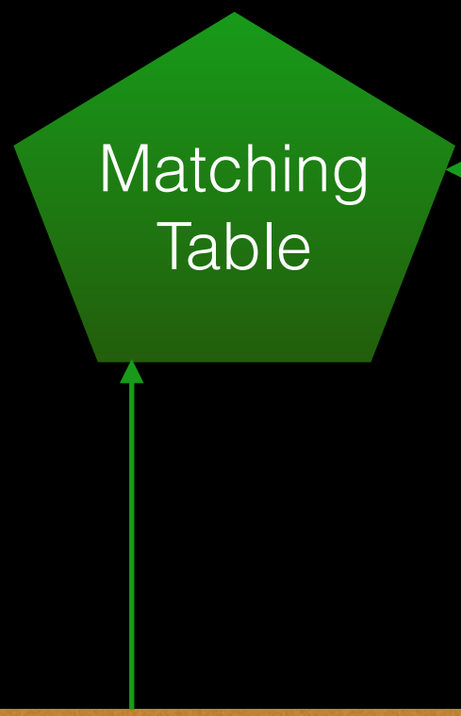
Stage 1 - Static Analysis

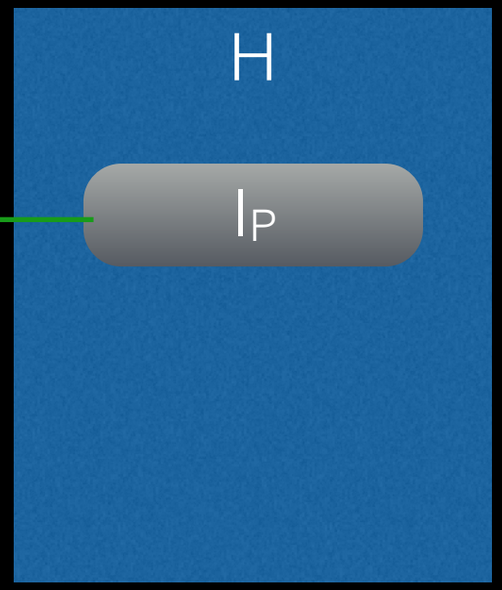


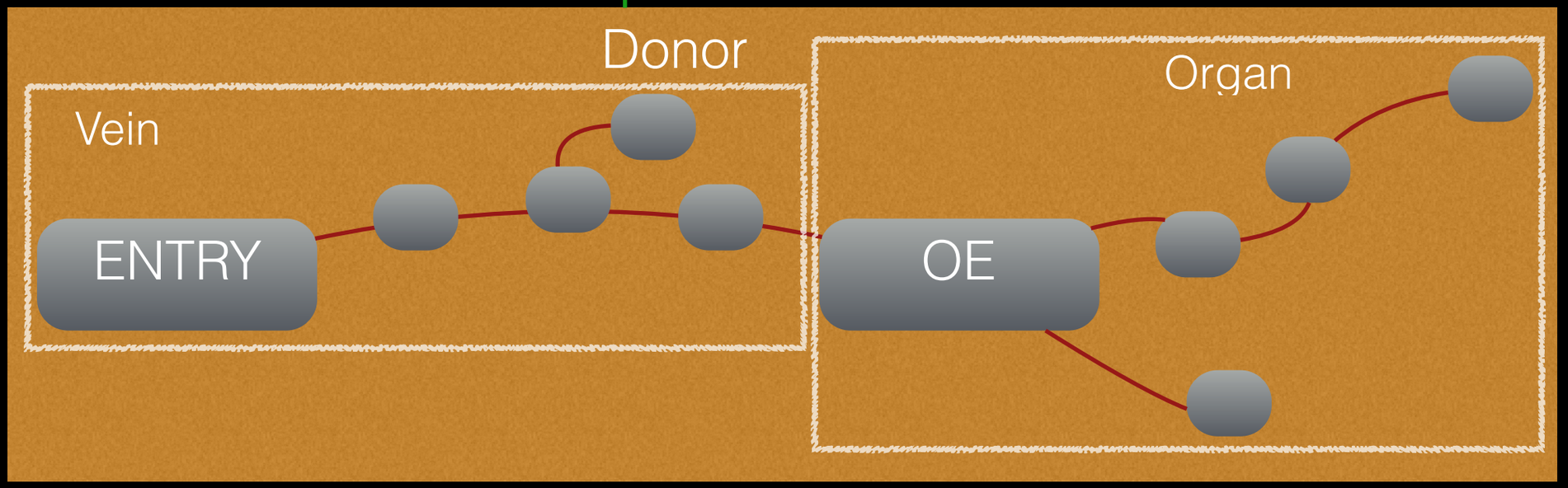


Stage 1 - Static Analysis

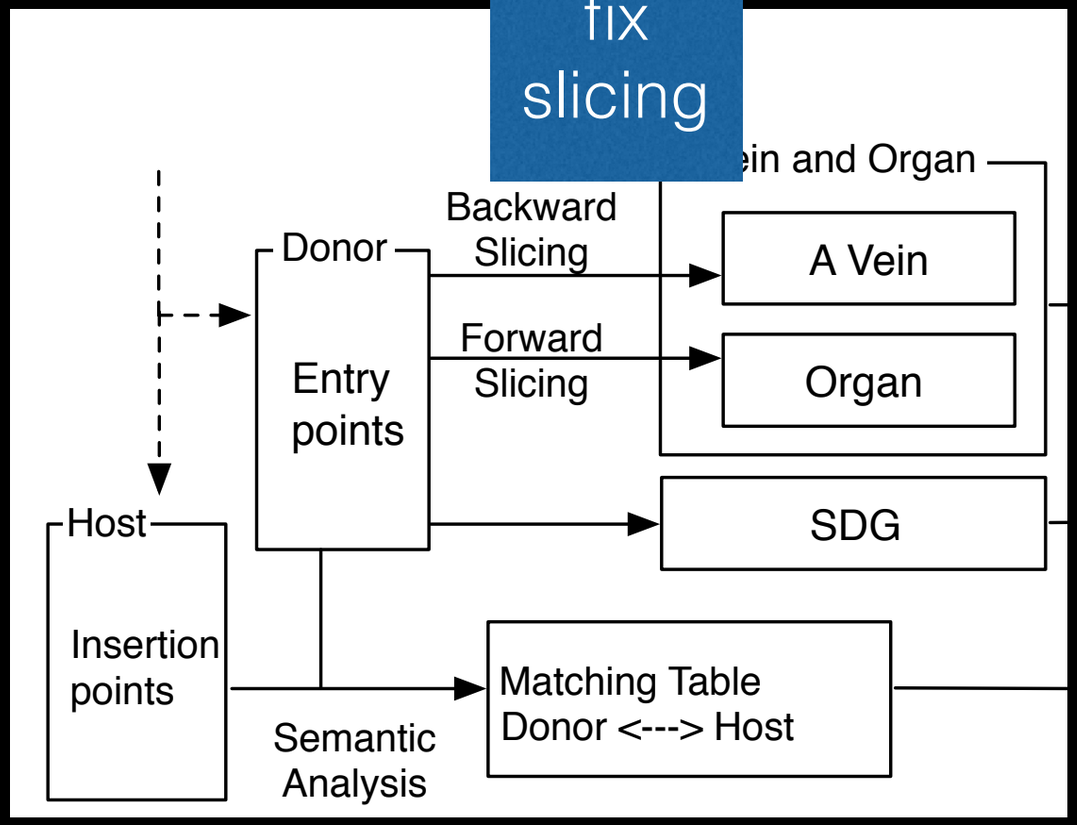


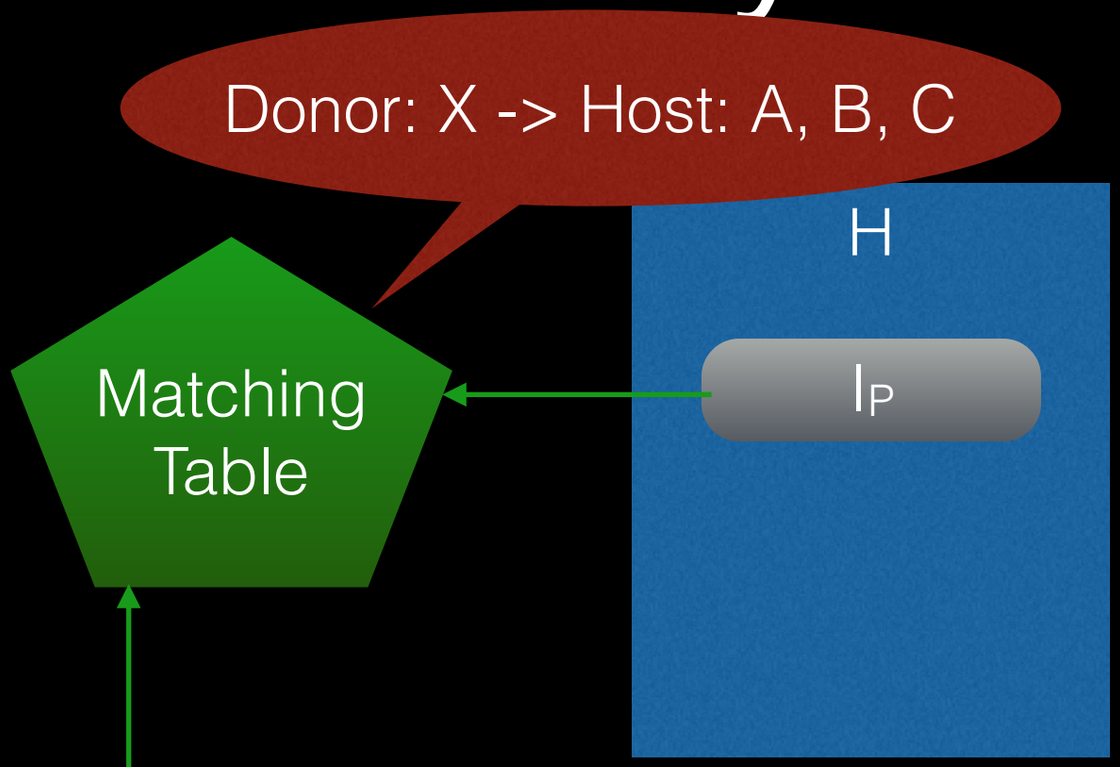


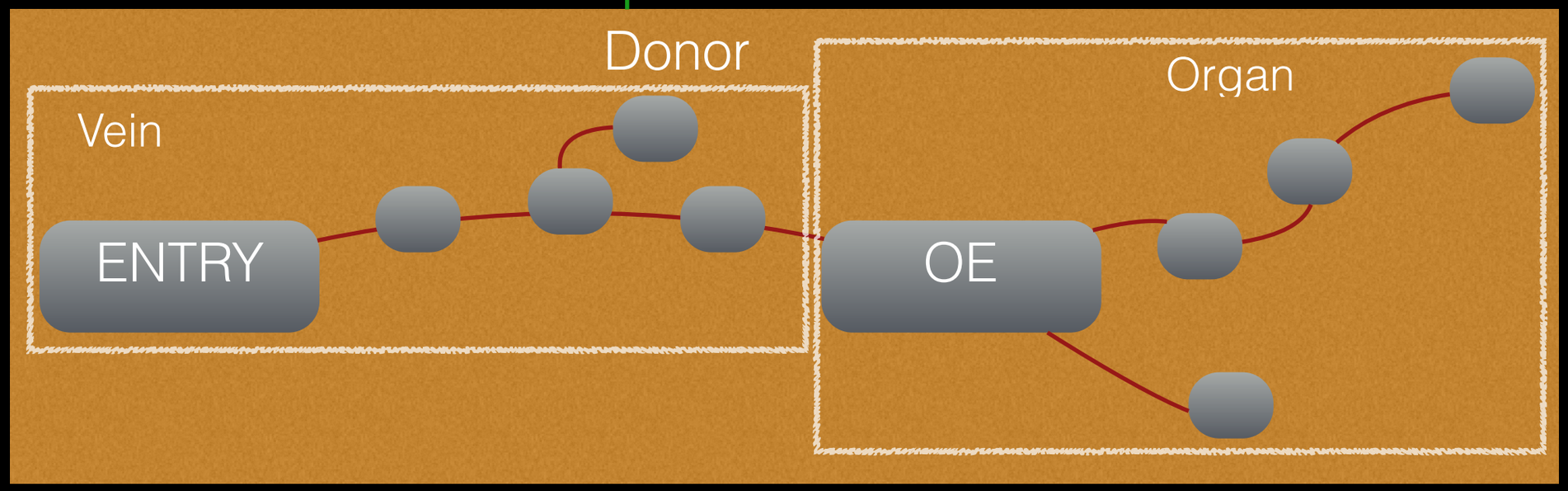




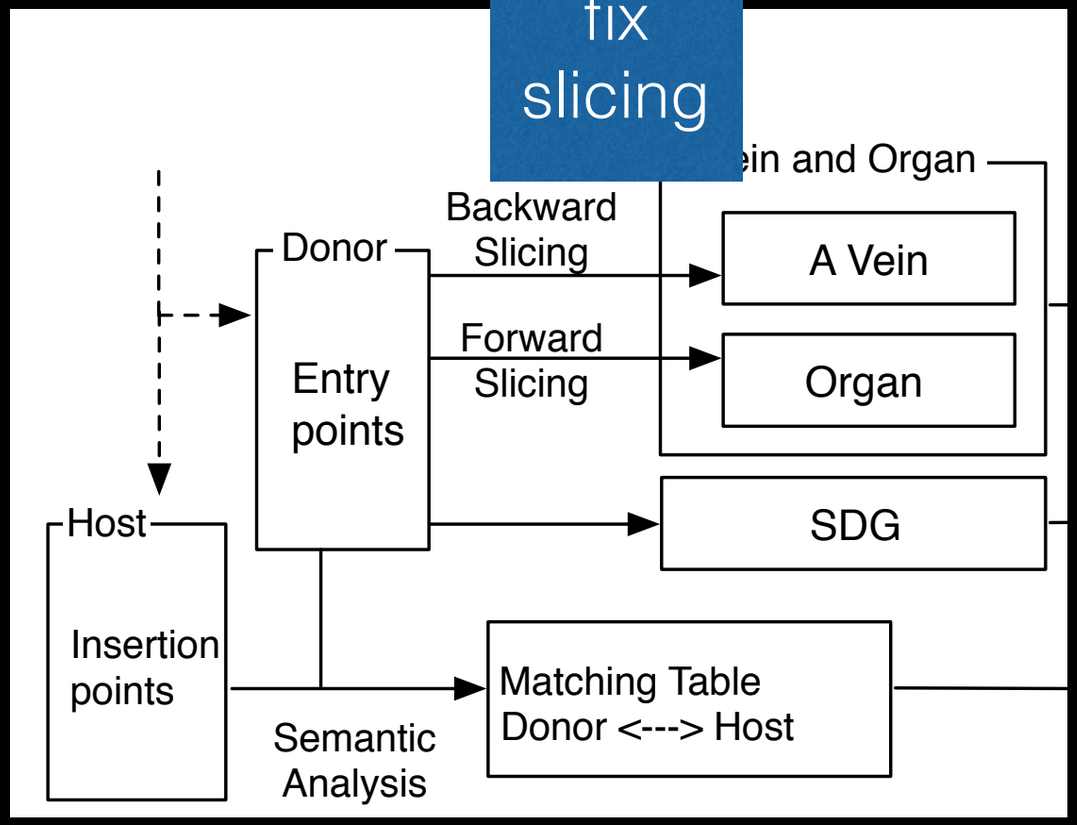
Stage 1 - Static Analysis

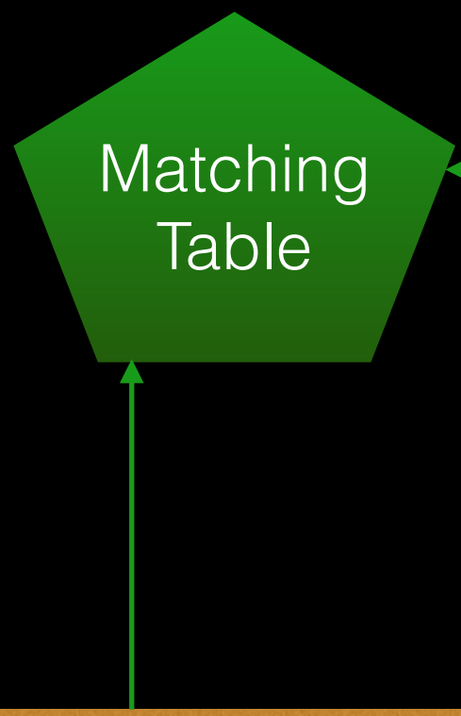


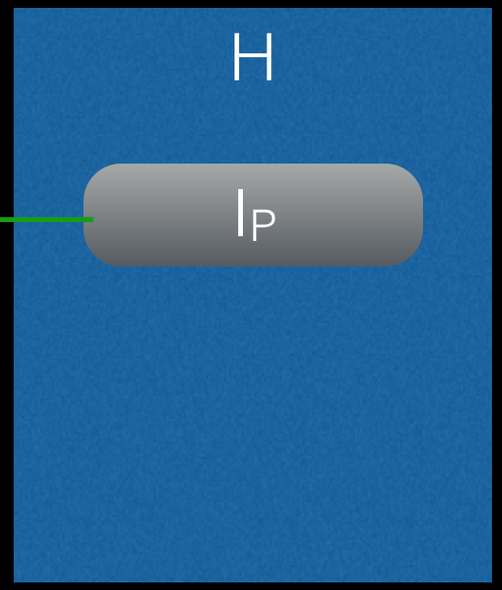


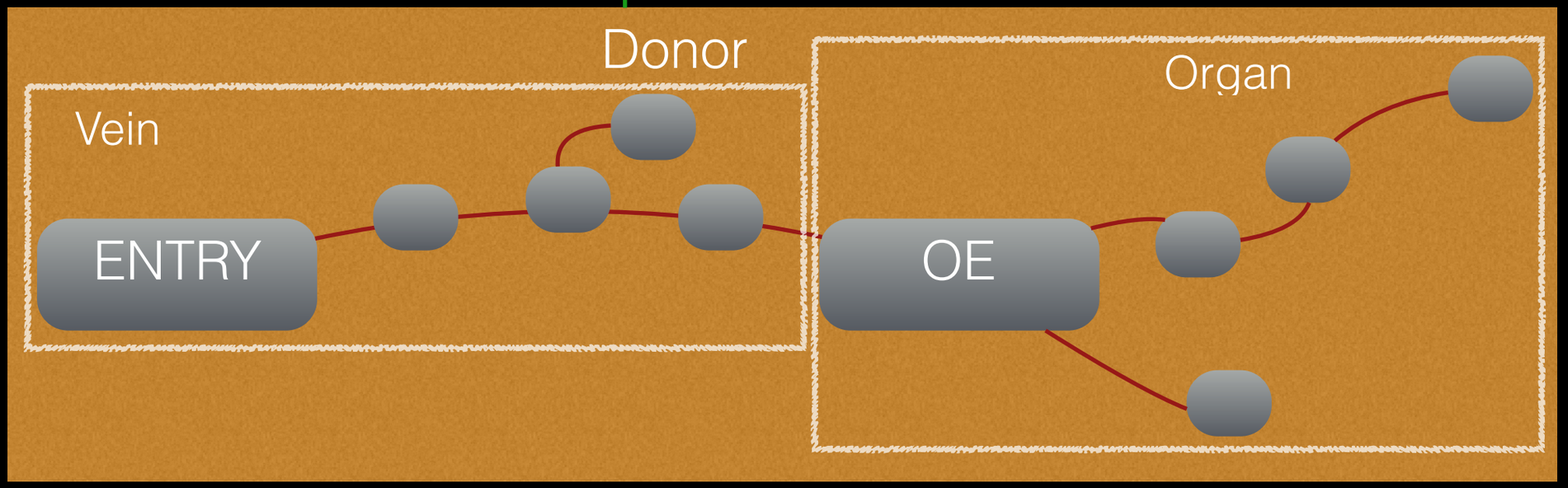


Stage 1 - Static Analysis

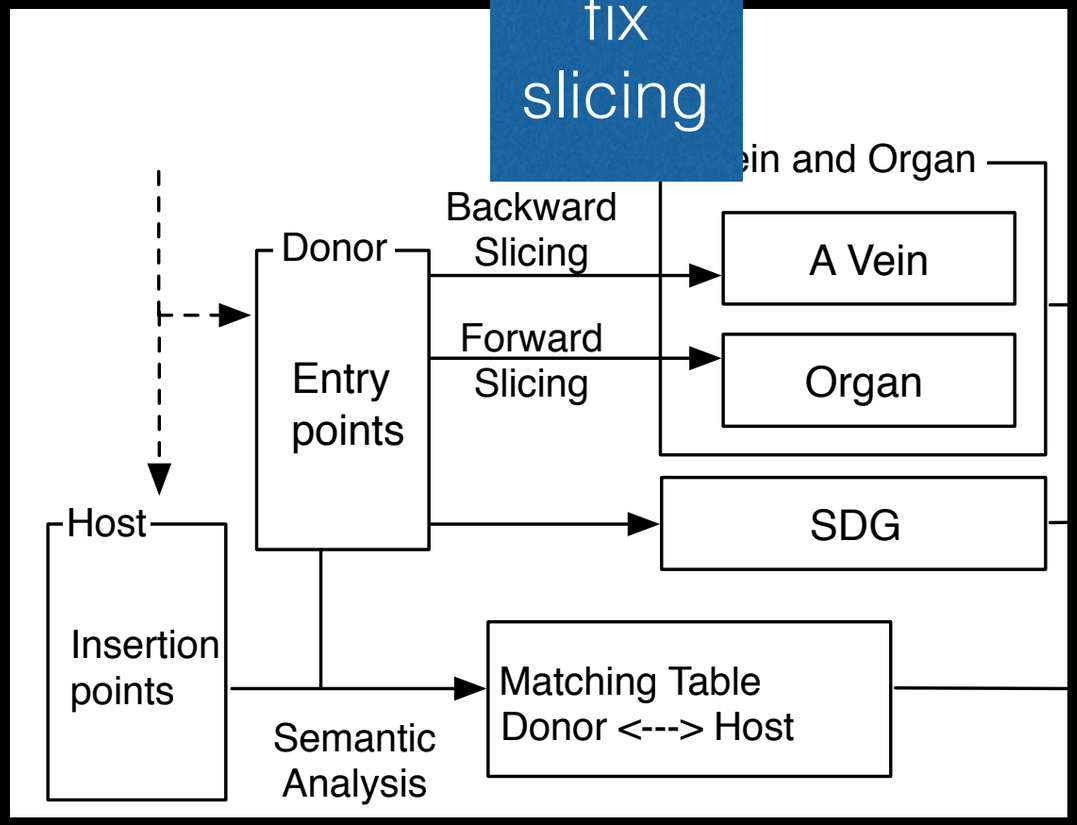


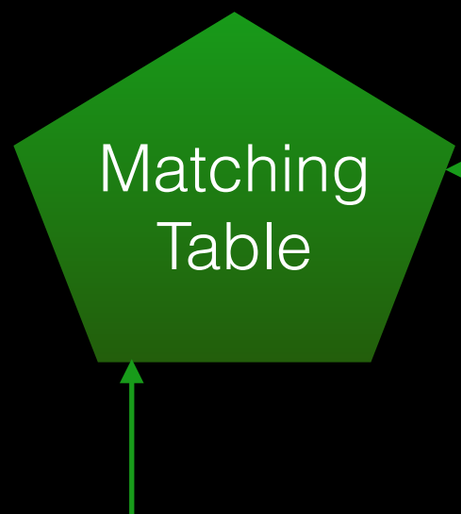


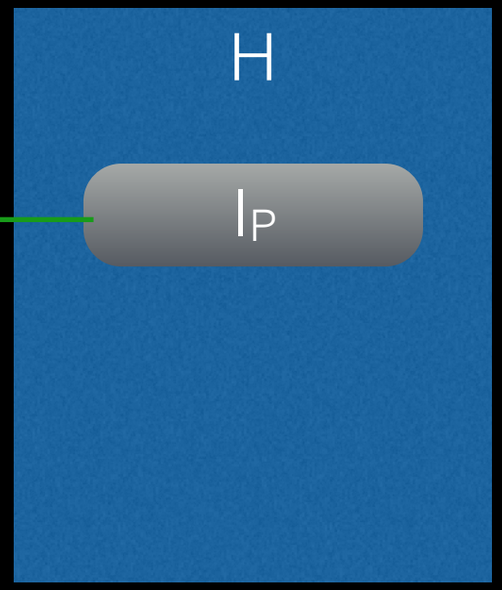


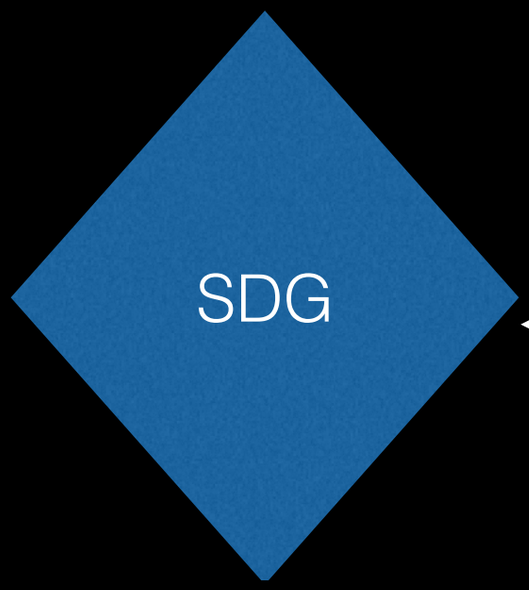


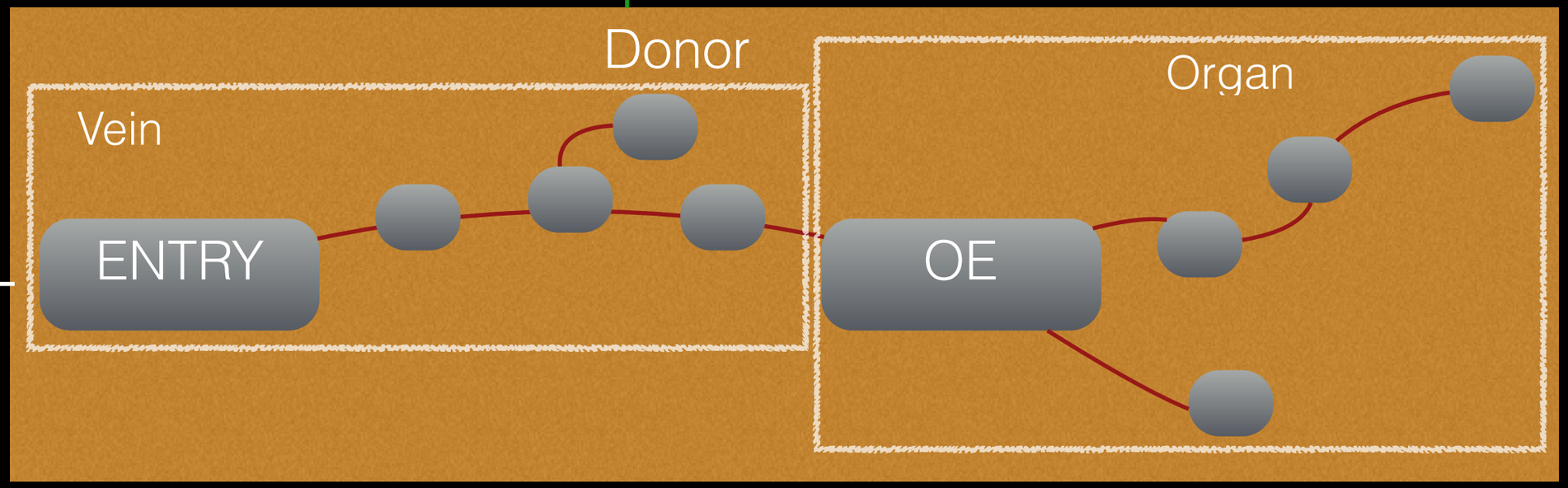
Stage 1 - Static Analysis





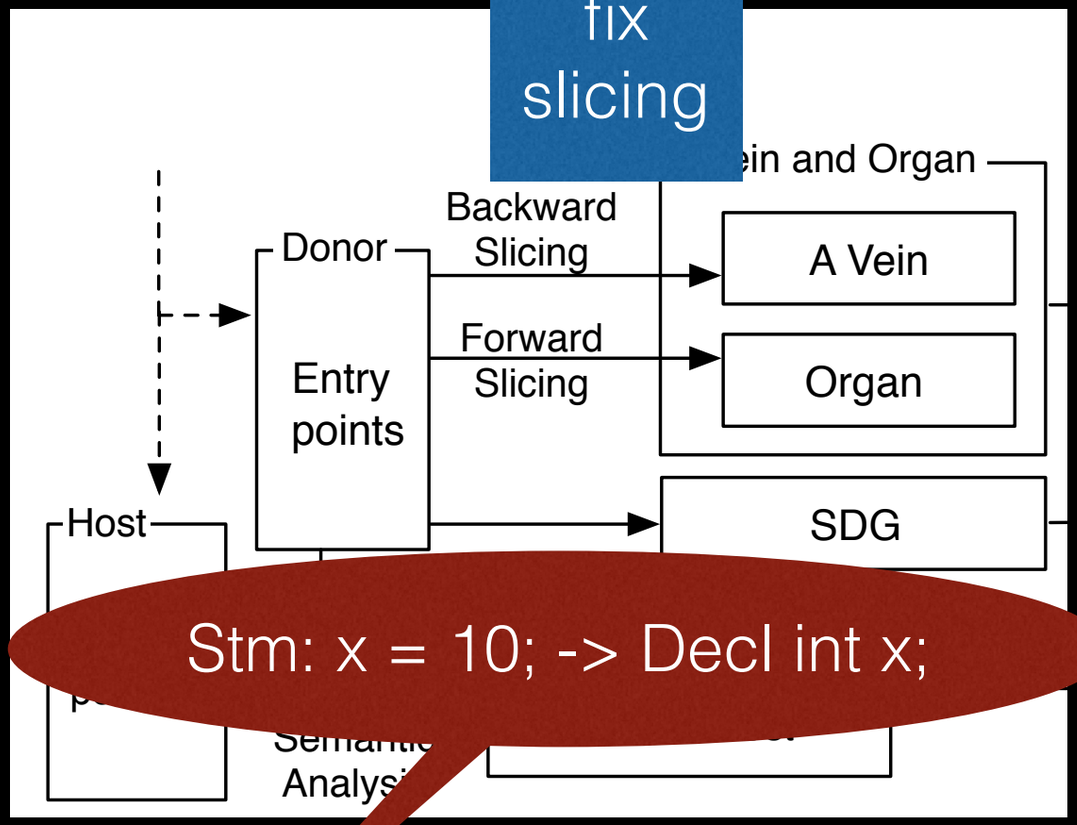






Stage 1 - Static Analysis

fix slicing

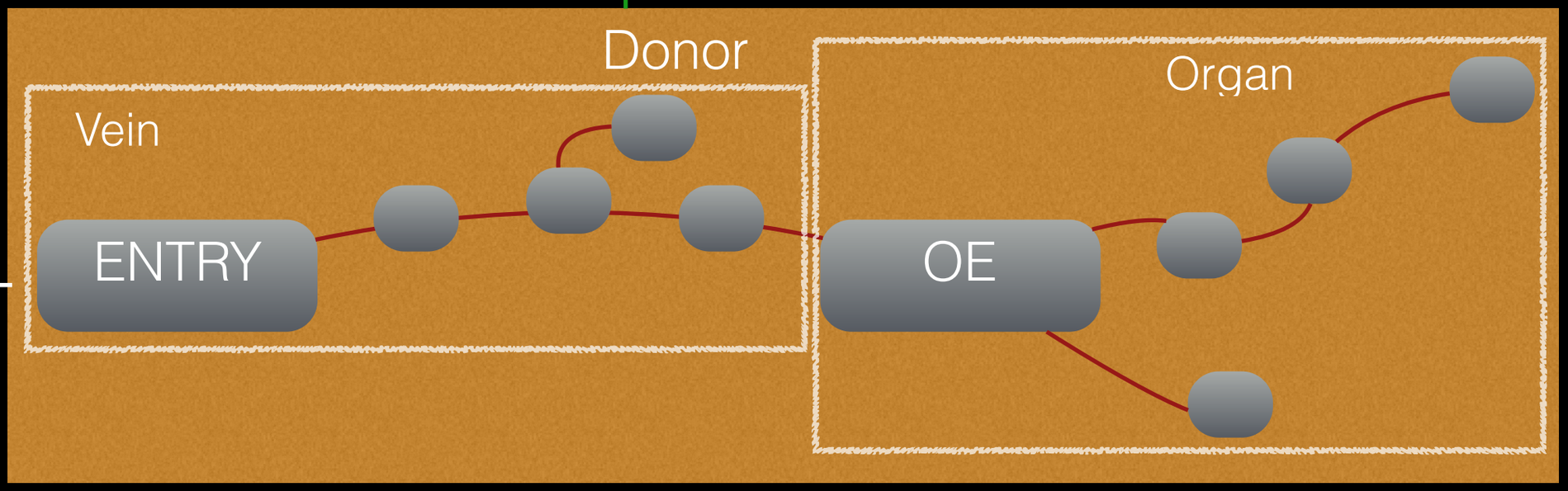


Matching Table

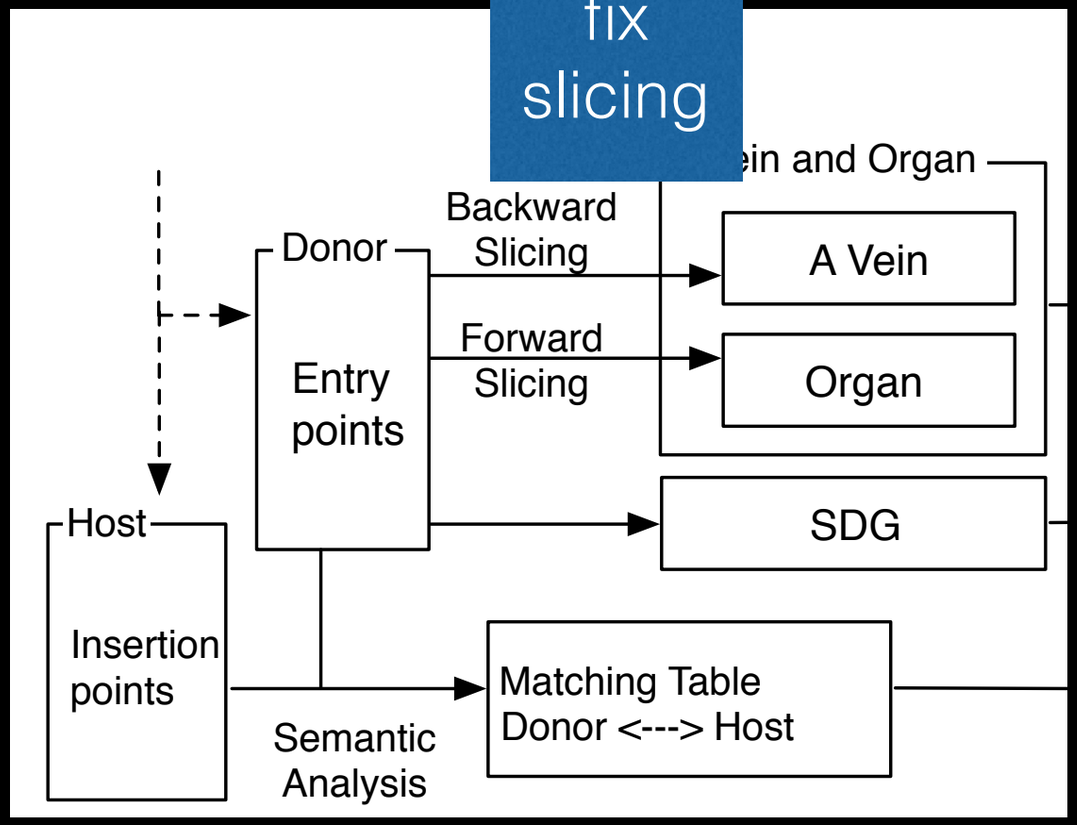
H

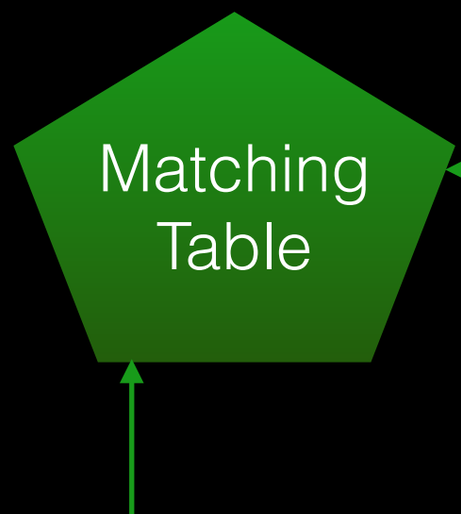
I_P

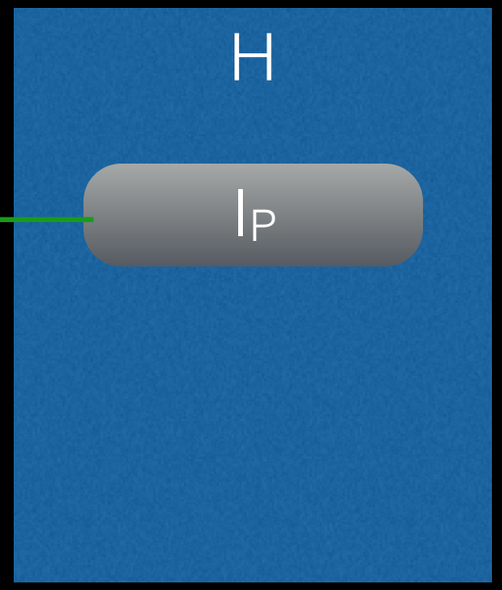
SDG

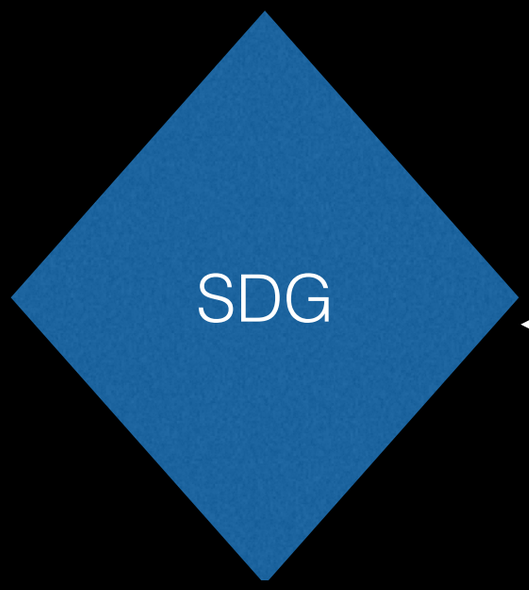


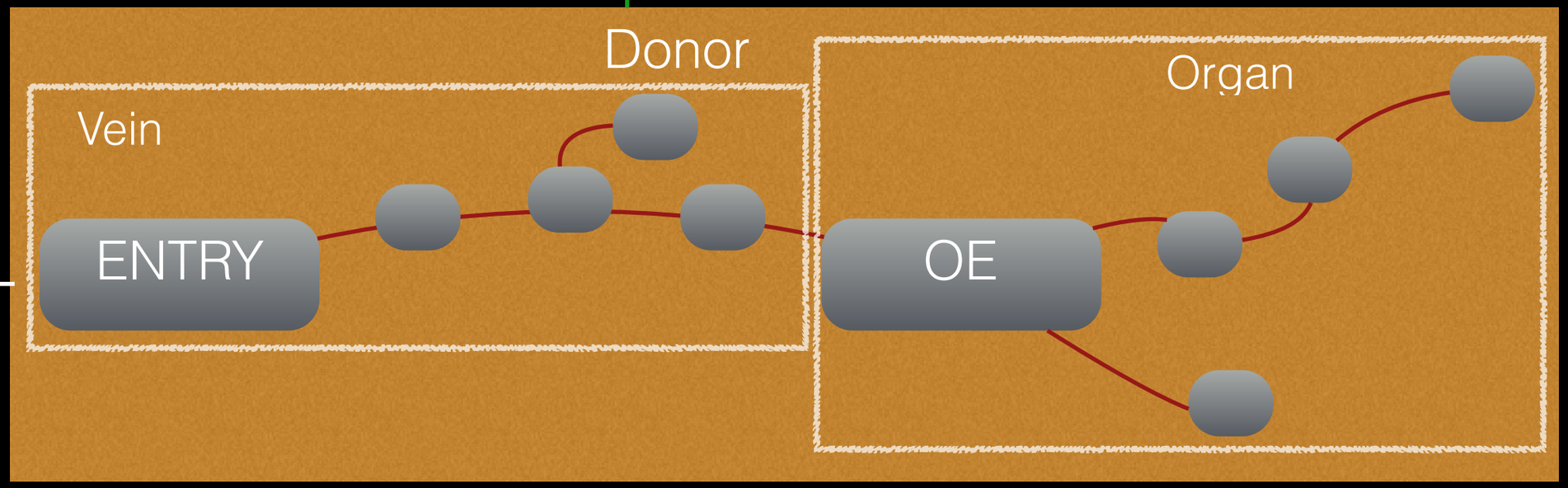
Stage 1 - Static Analysis



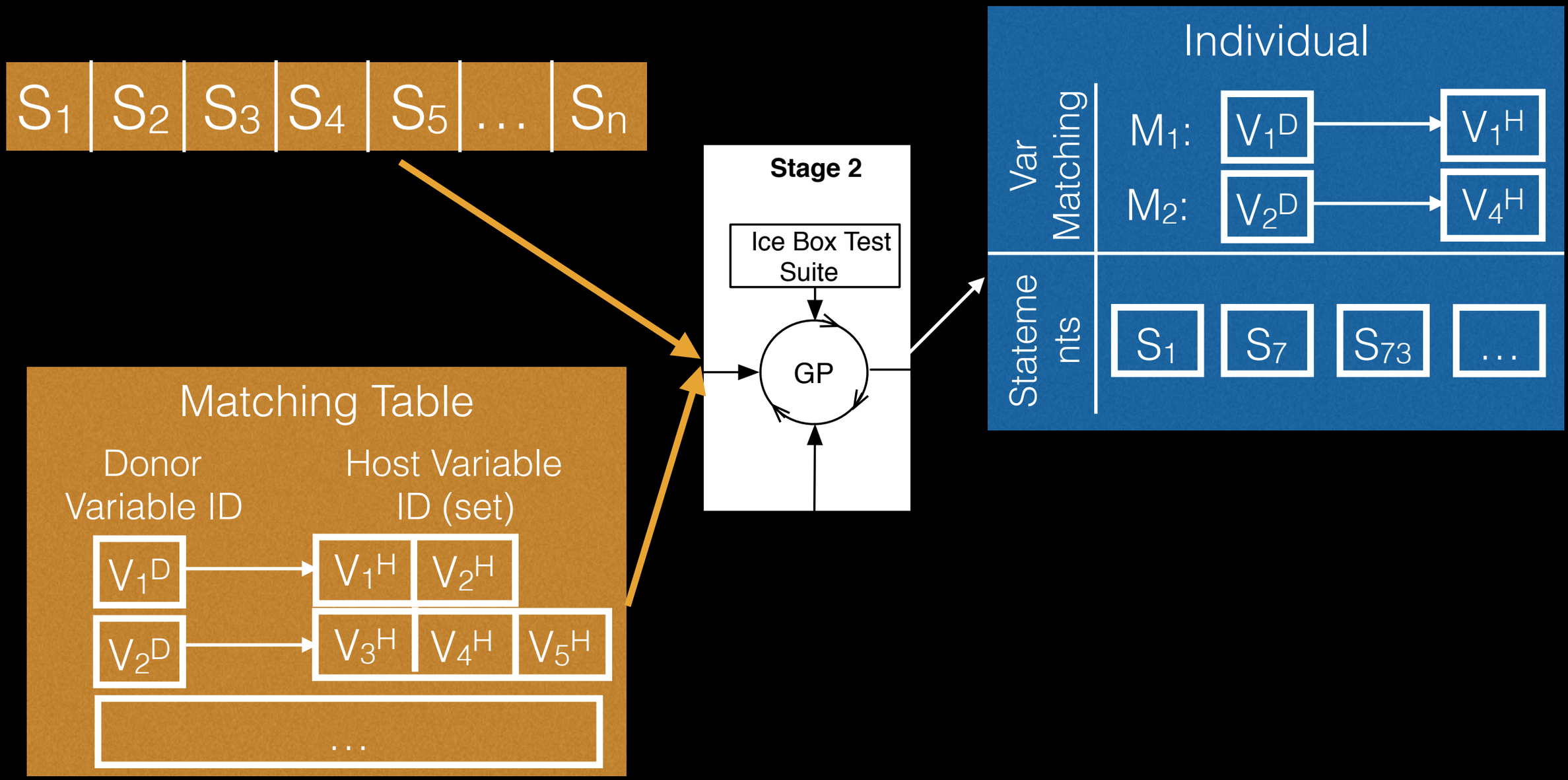




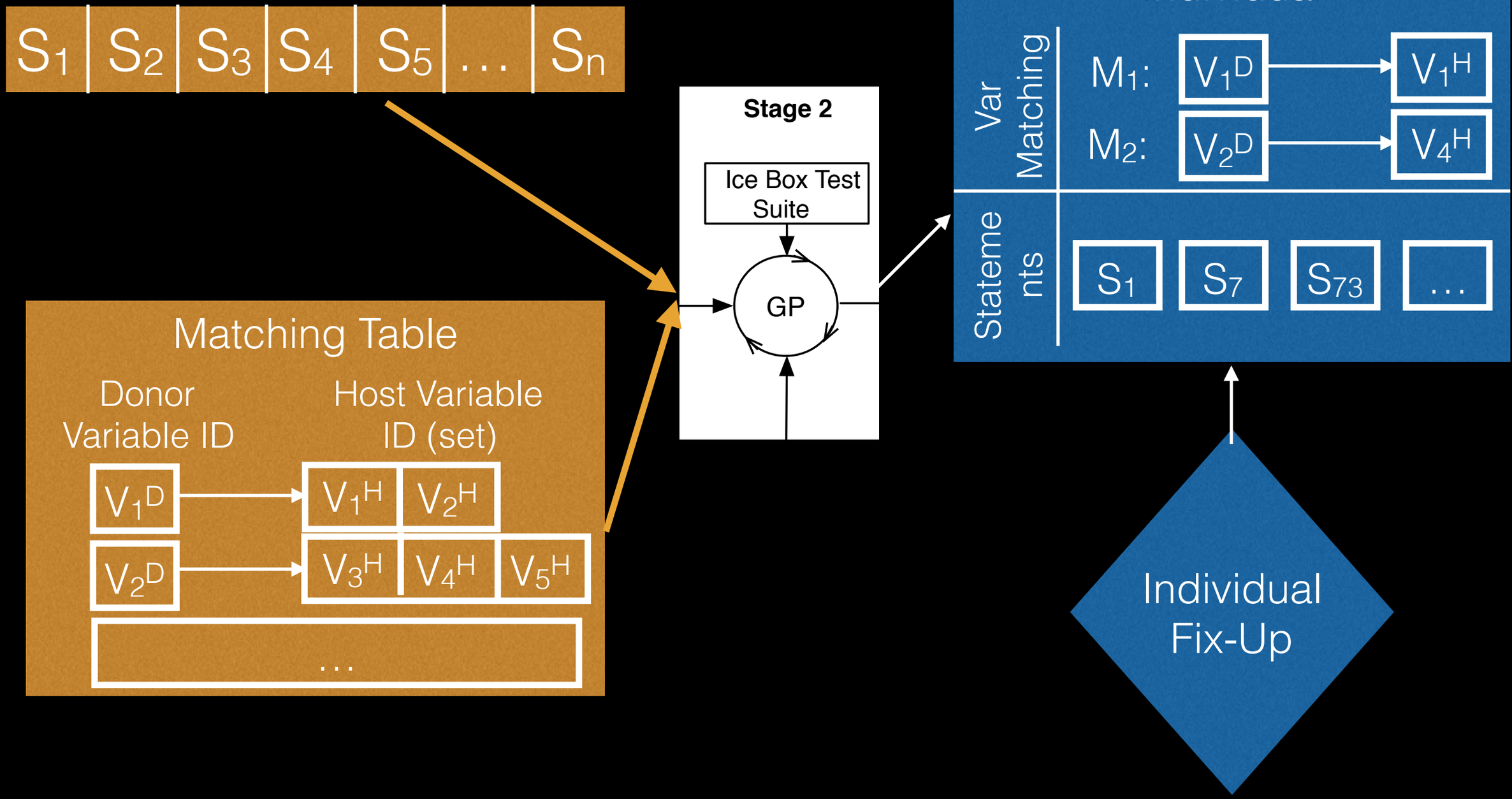




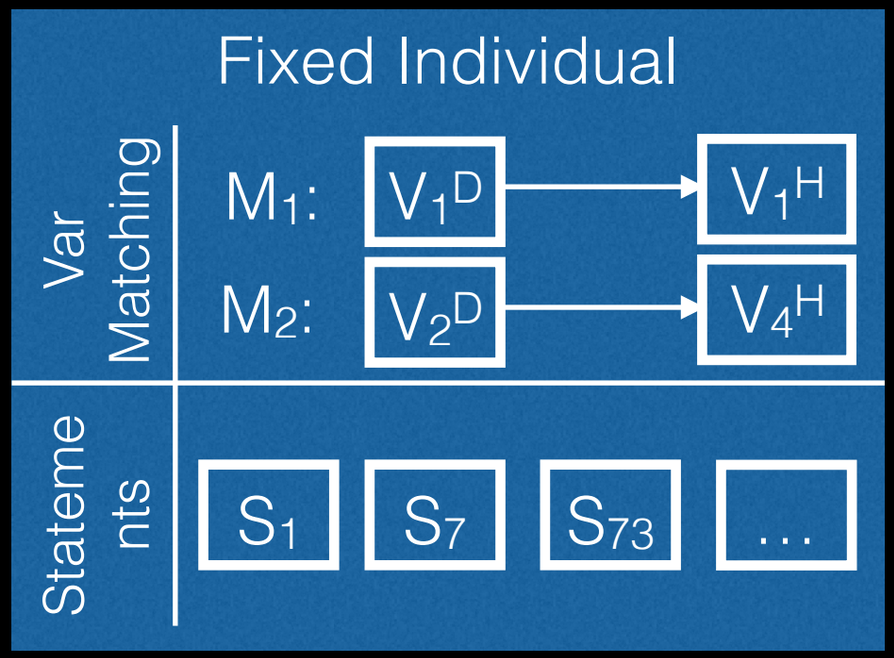
Stage 2 - Gp


S1 S2 S3 S4 S5 … Sn
Stage 2
Ice Box Test Suite
GP
Individual
Var Matching
M1: V1D V1H
M2: V2D V4H
Statements
S1 S7 S73 …
Matching Table
Donor Variable ID
Host Variable ID (set)
V1D V1H V2H
V2D V3H V4H V5H
…

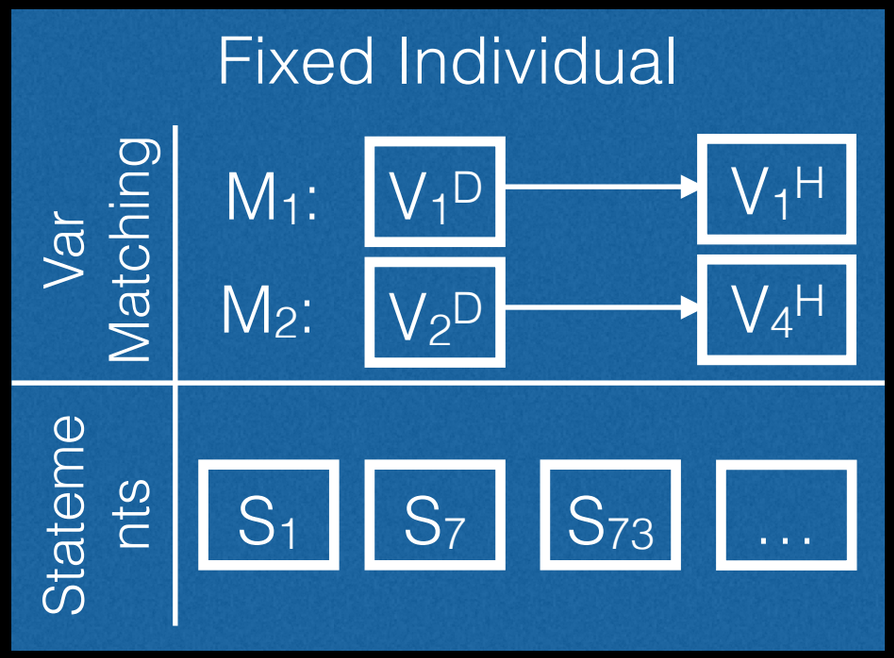
Stage 2 - Gp


S1 S2 S3 S4 S5 … Sn
Stage 2
Ice Box Test Suite
GP
Individual
Var Matching
M1: V1D V1H
M2: V2D V4H
Statements
S1 S7 S73 …
Matching Table
Donor Variable ID
Host Variable ID (set)
V1D V1H V2H
V2D V3H V4H V5H
…
Individual Fix-Up

Stage 2 - Fitness Evaluation

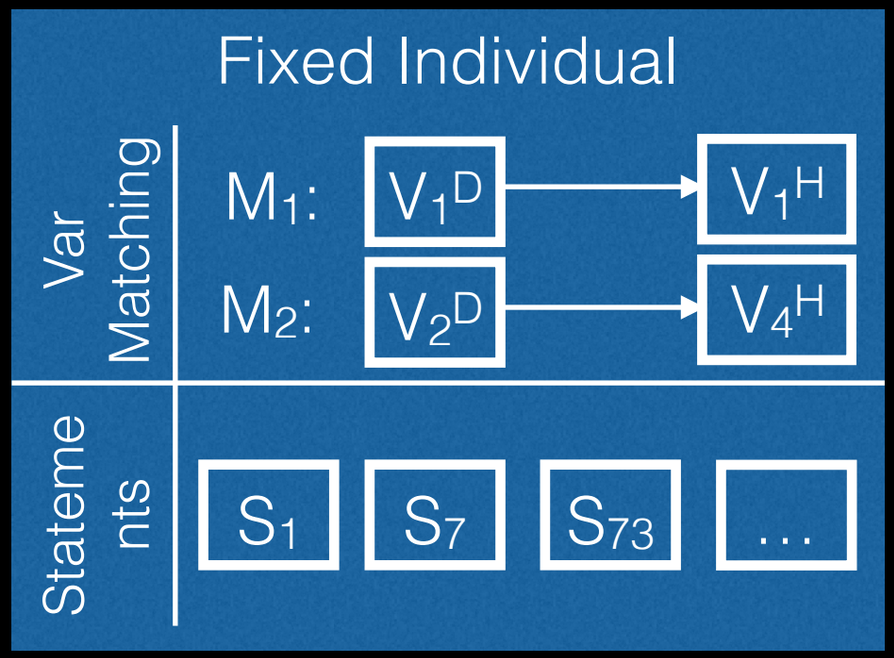


Stage 2 - Fitness Evaluation



$$fitness(i) = \begin{cases} \frac{1}{3}(1 + \frac{|TX_i|}{|T|} + \frac{|TP_i|}{|T|}) & i \in I_C \\ 0 & i \notin I_C \end{cases}$$

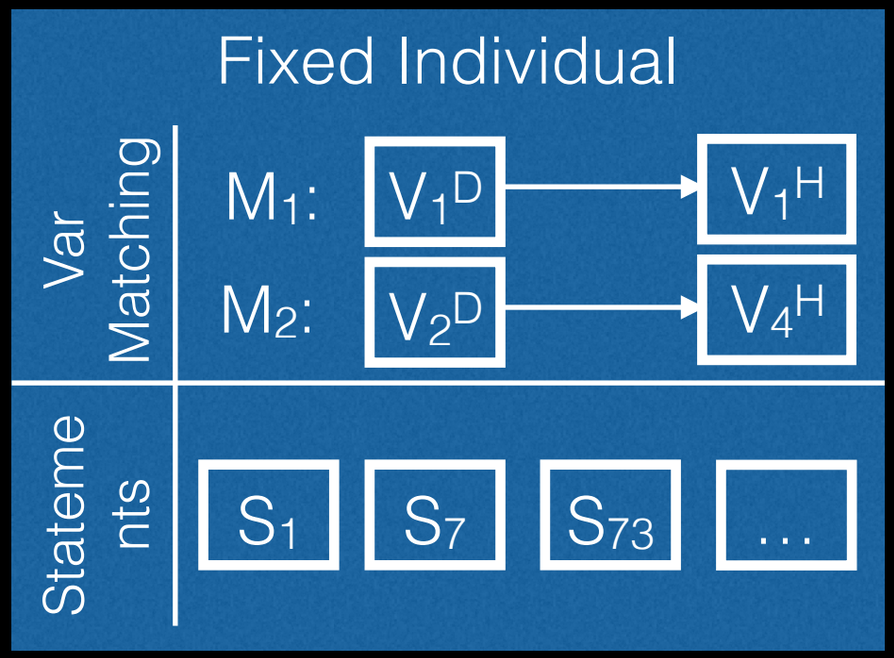
Stage 2 - Fitness Evaluation



Does it compile?

$$fitness(i) = \begin{cases} \frac{1}{3}(1 + \frac{|TX_i|}{|T|} + \frac{|TP_i|}{|T|}) & i \in I_C \\ 0 & i \notin I_C \end{cases}$$

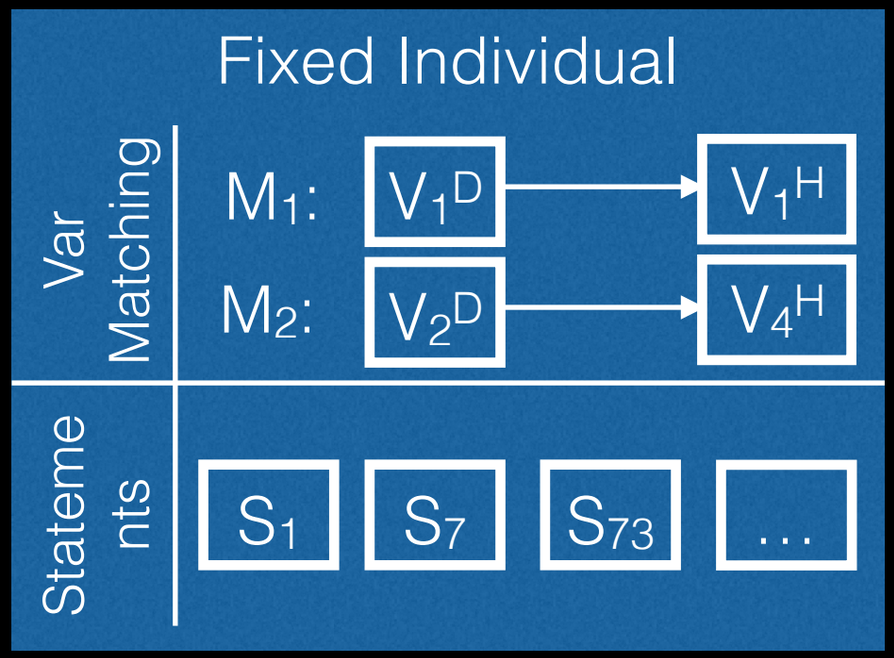
Stage 2 - Fitness Evaluation



$$fitness(i) = \begin{cases} \frac{1}{3}(1 + \frac{|TX_i|}{|T|} + \frac{|TP_i|}{|T|}) & i \in I_C \\ 0 & i \notin I_C \end{cases}$$

Week Proxies:
Does it executes test cases without crashing?

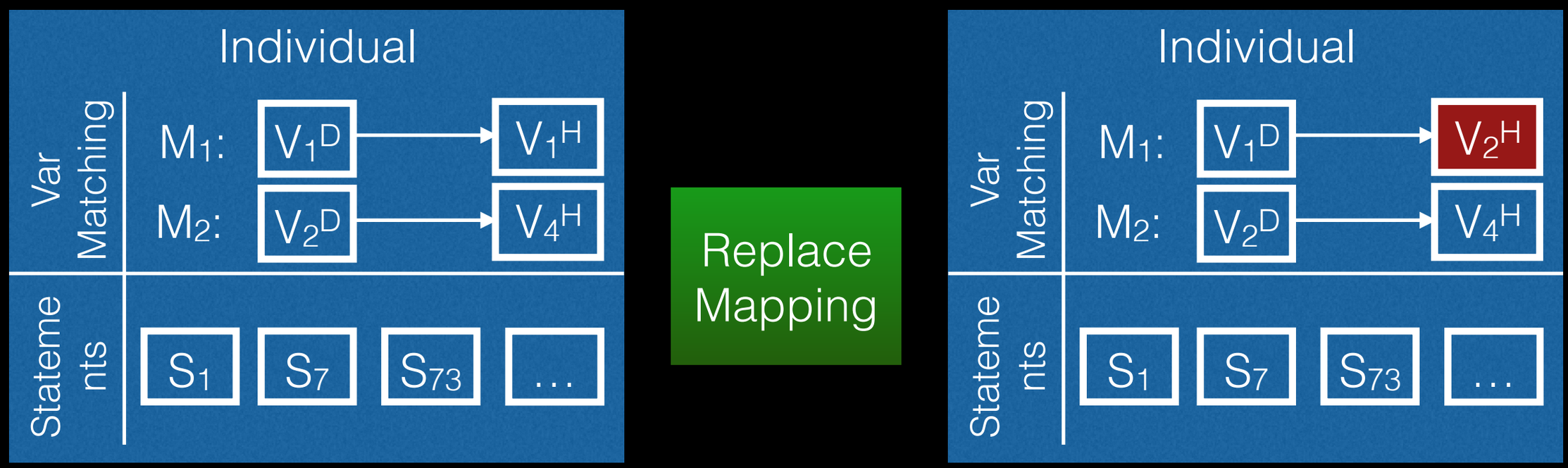
Stage 2 - Fitness Evaluation



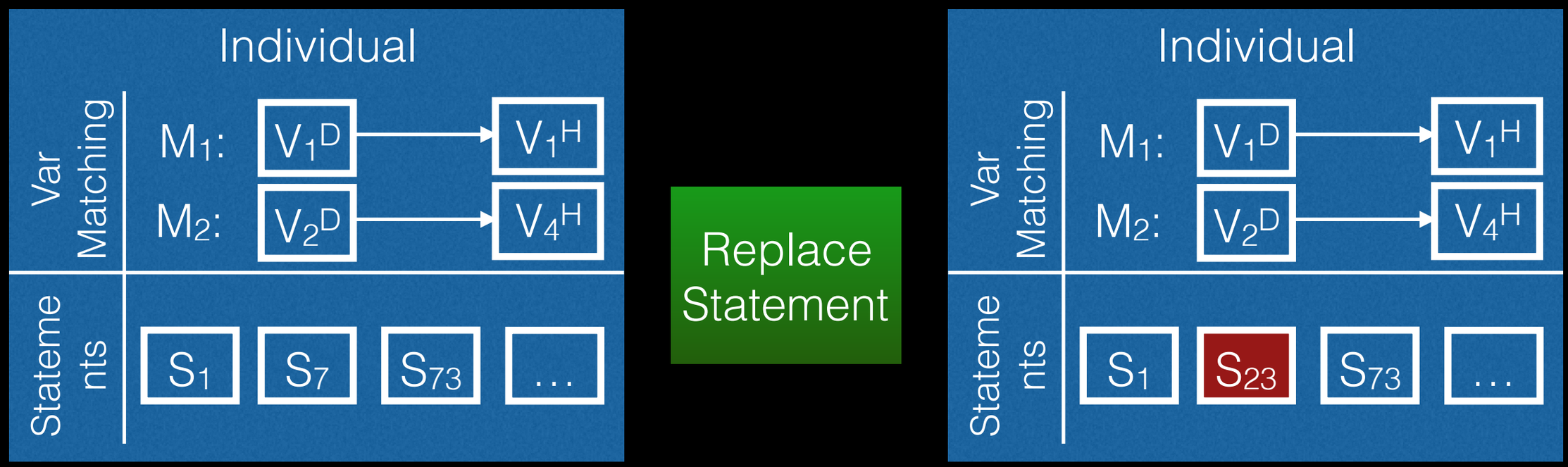
$$fitness(i) = \begin{cases} \frac{1}{3}(1 + \frac{|TX_i|}{|T|} + \frac{|TP_i|}{|T|}) & i \in I_C \\ 0 & i \notin I_C \end{cases}$$

Strong Proxies: Does it produce the correct output?

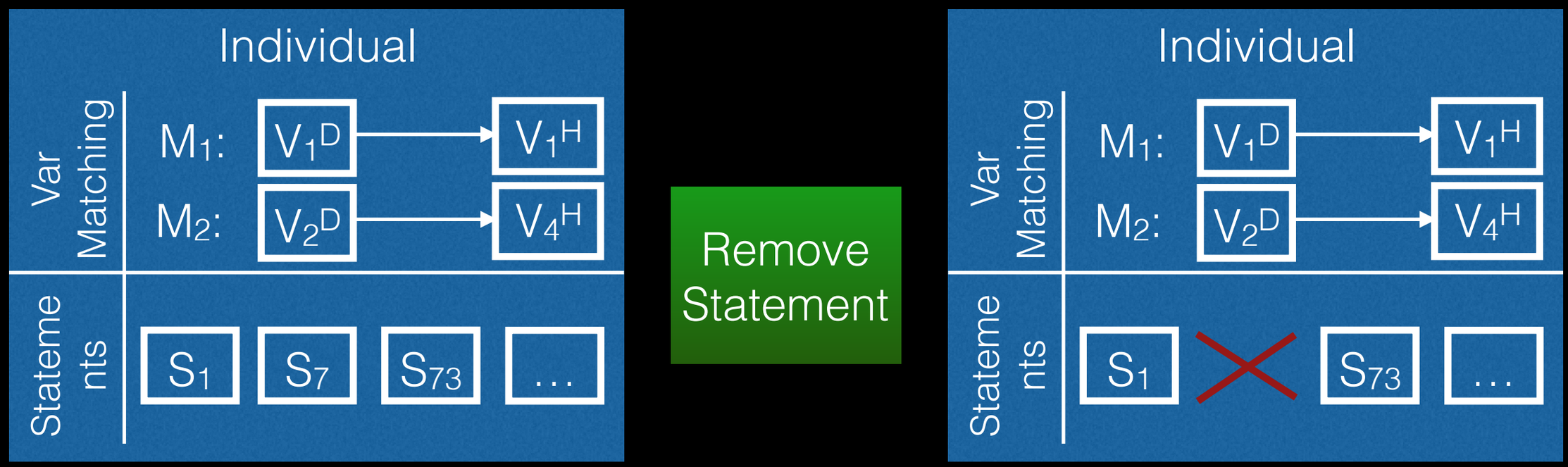
Stage 2 - Gp Operators



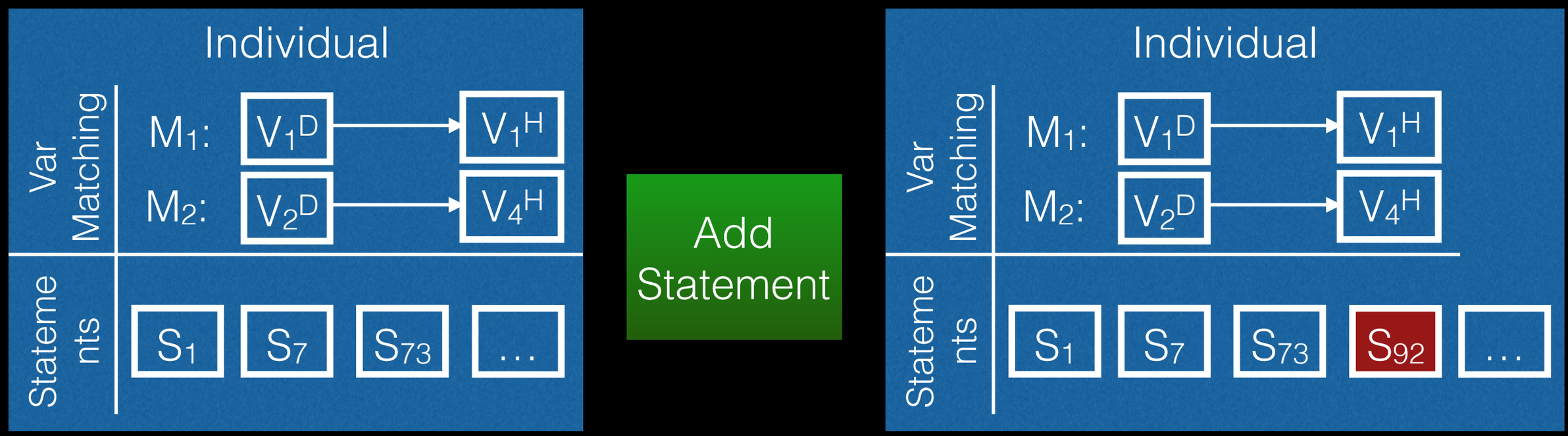
Stage 2 - Gp Operators



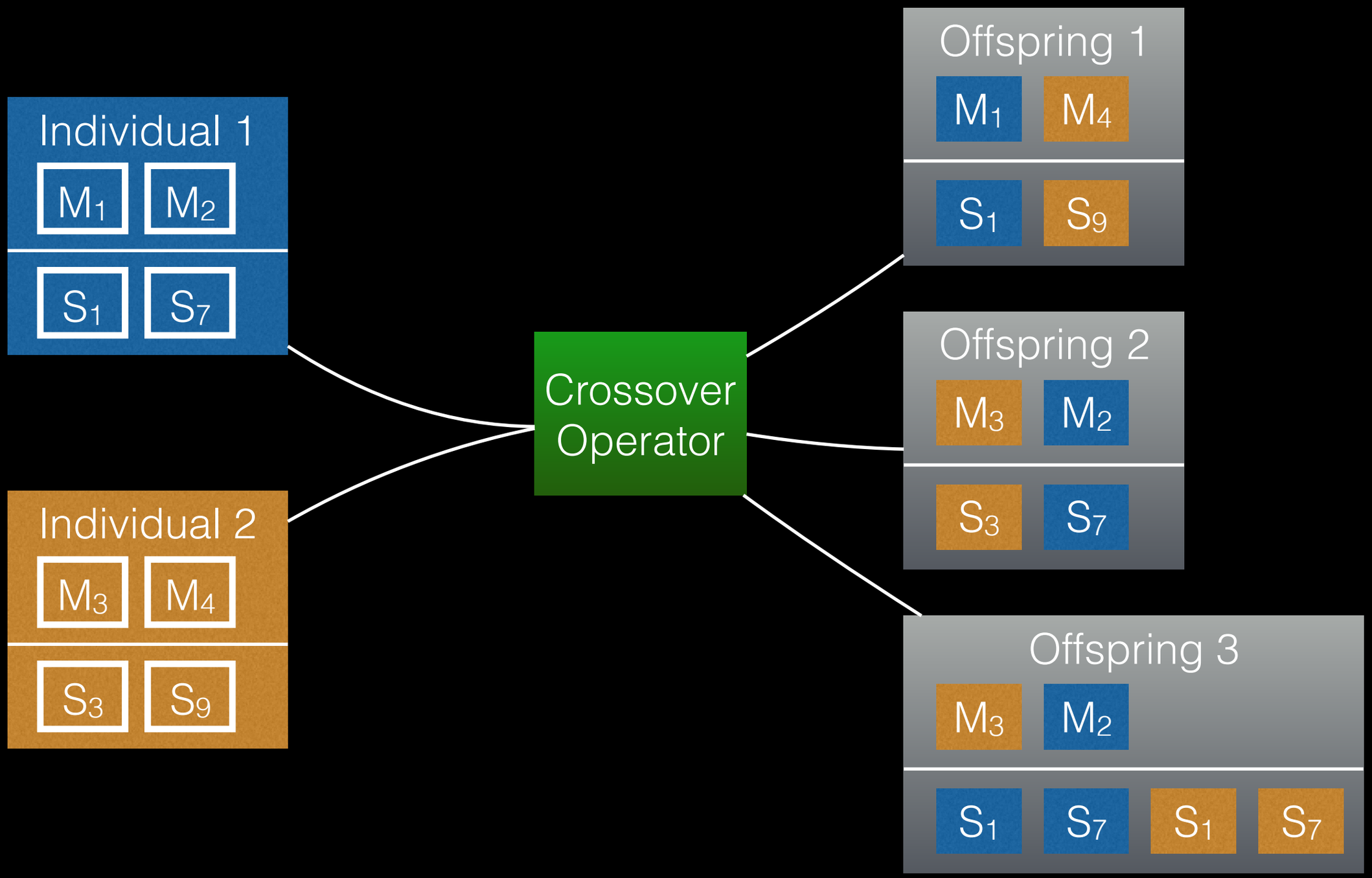
Stage 2 - Gp Operators



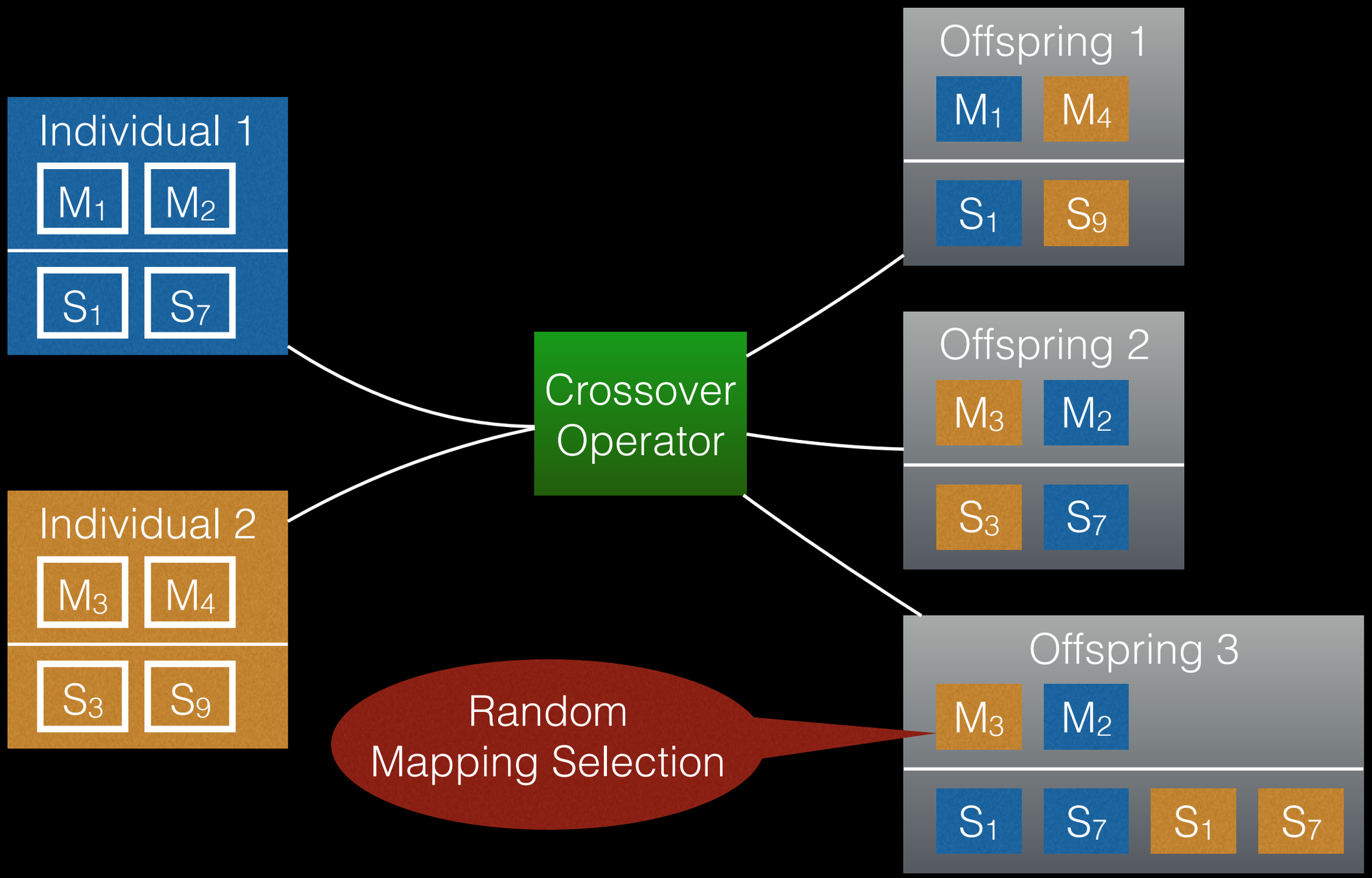
Stage 2 - Gp Operators



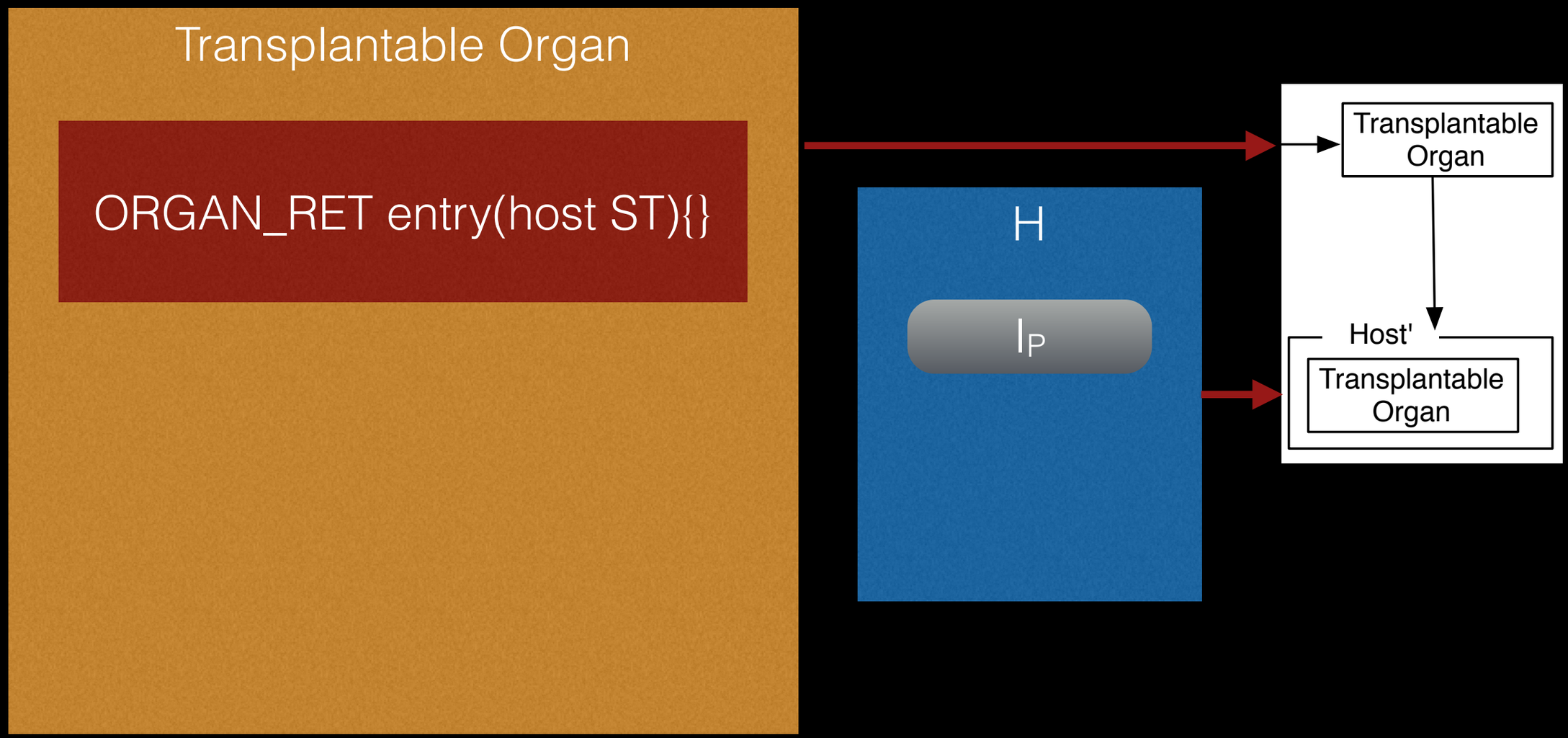
Stage 2 - Gp Operators



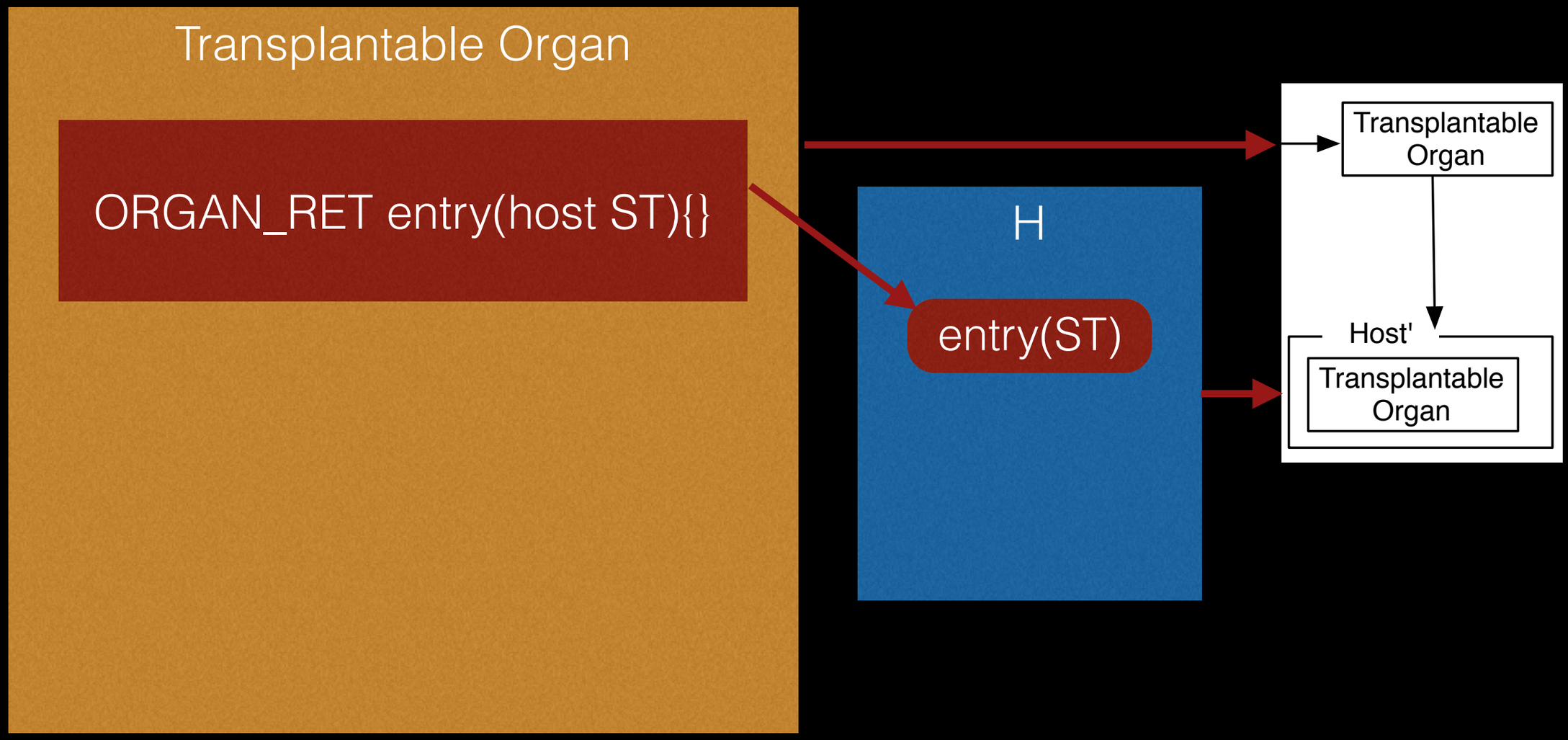
Stage 2 - Gp Operators



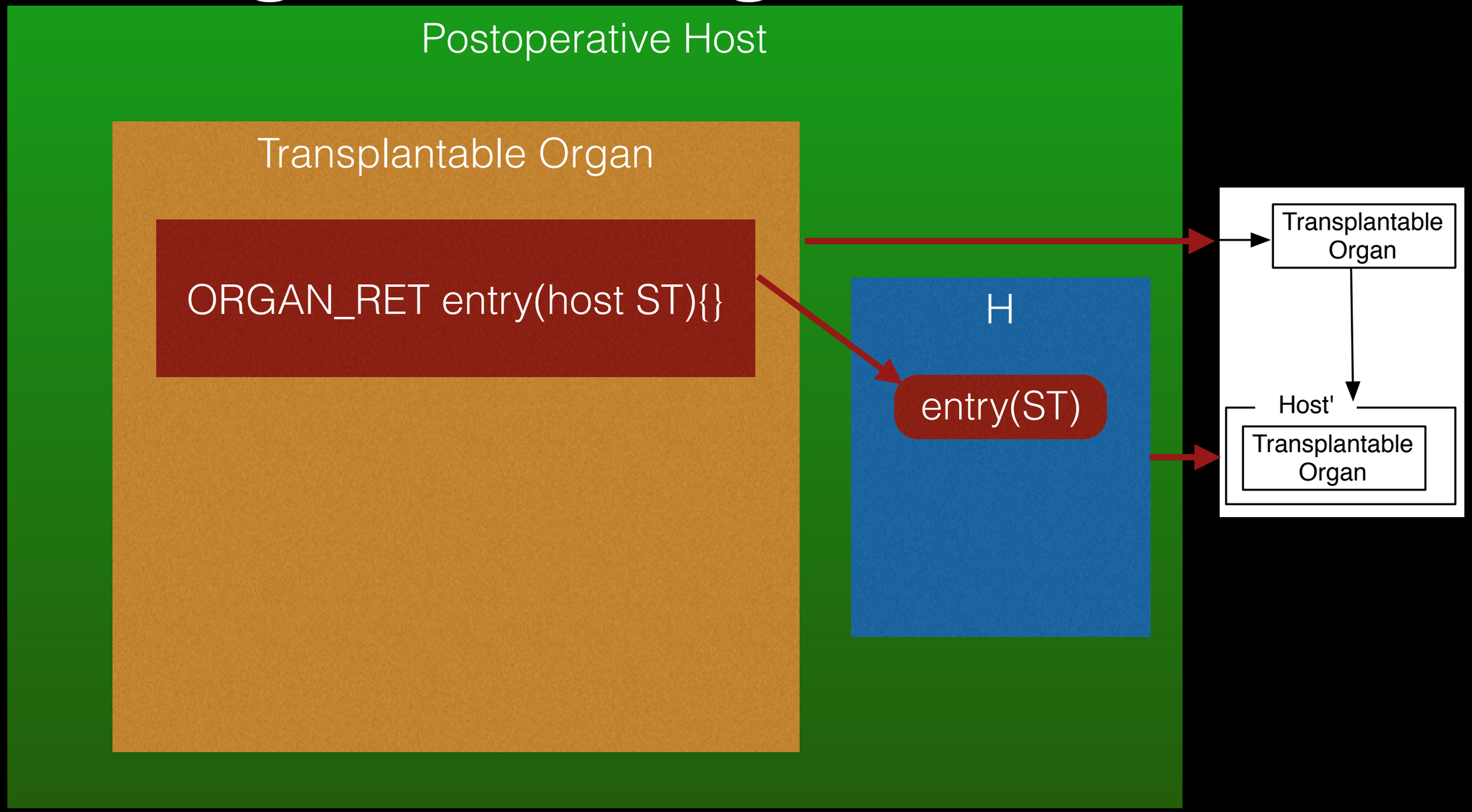
Stage 3 - Organ Insertion



Stage 3 - Organ Insertion



Stage 3 - Organ Insertion



Demo

Validation

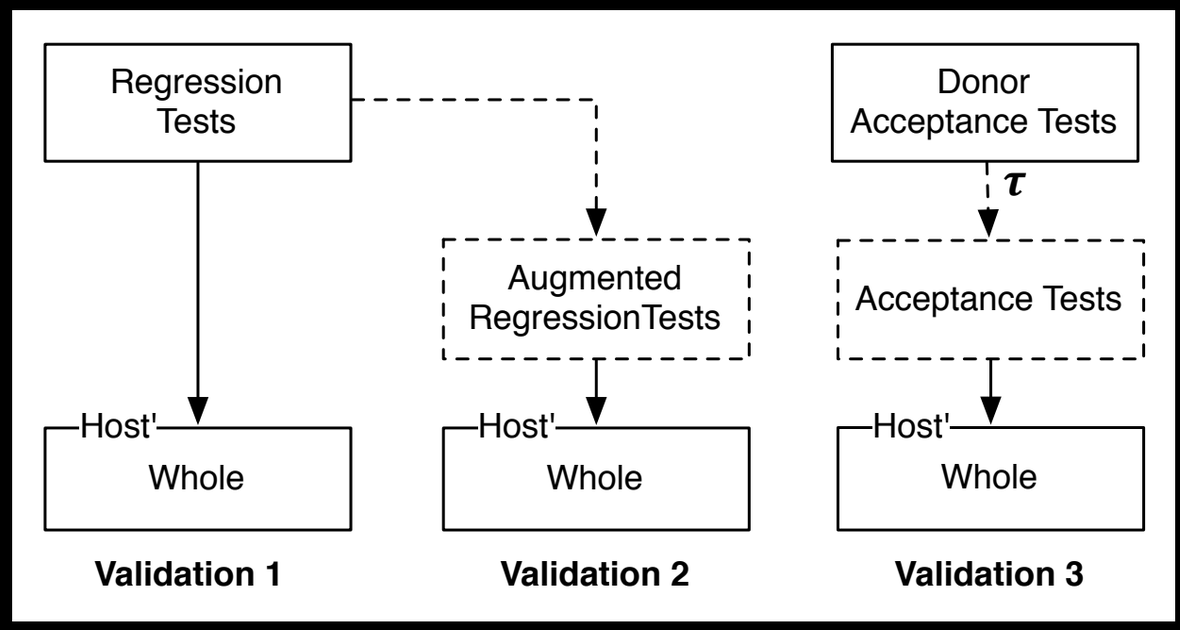
4 validation steps:

Regression Tests;

Augmented Regression Tests;

Acceptance Tests;

Manual Validation;



Subject Programs

Subjects	Type	Size	Regr.	Isolation
Idct	Donor	2.3k	-	3-5
Mytar	Donor	0.4k	-	4
Cflow	Donor	25k	-	6-20
Webserver	Donor	1.7k	-	3
TuxCrypt	Donor	2.7k	-	4-5
Pidgin	Host	363k	88	-
Cflow	Host	25k	21	-
SoX	Host	43k	157	-
Case Study				
x264	Donor	63k	-	1
GNU Indent	Donor	26k	-	6
GNU cflow	Donor	25k	-	8
Kate	Host	43k	42	-
VLC	Host	422k	27	-

Results

Case Studies

		All	Test Suites			Time
Donor	Host	Passed	Regression	Regression++	Acceptance	Avarage
x264	VLC	1	1	1	1	26 (hours)
cflow	Kate	16	20*	17	18	101
Indent	Kate	18	20*	18	19	31

Empirical Study

		All	Test Suites		
Donor	Host	Passed	Regression	Regression++	Acceptance
Idct	Pidgin	16	20	17	16
Mytar	Pidgin	16	20	18	20
Web	Pidgin	0	20	0	18
Cflow	Pidgin	15	20	15	16
Tux	Pidgin	15	20	17	16
Idct	Cflow	16	17	16	16
Mytar	Cflow	17	17	17	20
Web	Cflow	0	0	0	17
Cflow	Cflow	20	20	20	20
Tux	Cflow	14	15	14	16
Idct	SoX	15	18	17	16
Mytar	SoX	17	17	17	20
Web	SoX	0	0	0	17
Cflow	SoX	14	16	15	14
Tux	SoX	13	13	13	14
TOTAL		188/300	233/300	196/300	256/300

Publication

- Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In: ISSTA (2015), to appear
- Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation - Artifact Evaluation. In: ISSTA15-AE , accepted
- Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated Transplantation of Call Graph and Layout Features into Kate. In: *SSBSE* (2015), to appear
- Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: μSCALPEL, at http://crest.cs.ucl. ac.uk/autotransplantation/MuScalpel.html

Publication

Automated Software Transplantation

Earl T. Barr Mark Harman Yue Jia Alexandru Marginean Justyna Petke

CREST, University College London, Malet Place, London, WC1E 6BT, UK
{e.barr,m.harman,yue.jia,alexandru.marginean.13,j.petke}@ucl.ac.uk

ABSTRACT

Automated transplantation would open many exciting avenues for software development: suppose we could autotransplant code from one system into another, entirely unrelated, system. This paper introduces a theory, an algorithm, and a tool that achieve this. Leveraging lightweight annotation, program analysis identifies an organ (interesting behavior to transplant); testing validates that the organ exhibits the desired behavior during its extraction and after its implantation into a host. While we do not claim automated transplantation is now a solved problem, our results are encouraging: we report that in 12 of 15 experiments, involving 5 donors and 3 hosts (all popular real-world systems), we successfully autotransplanted new functionality and passed all regression tests. Autotransplantation is also already useful: in 26 hours computation time we successfully autotransplanted the H.264 video encoding functionality from the x264 system to the VLC media player; compare this to upgrading x264 within VLC, a task that we estimate, from VLC's version history, took human programmers an average of 20 days of elapsed, as opposed to dedicated, time.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Automated software transplantation, autotransplantation, genetic improvement

1. INTRODUCTION

Software engineers spend a great deal of time extracting, porting, and rewriting existing code to extend the functionality of existing software systems. Currently, this is tedious, laborious, and slow [15]. The research community has provided analyses and partial support for such manual code reuse: for example, clone detection [5, 32, 36, 60], code migration [38, 58], code salvaging [12], reuse [13, 14, 16], dependence analysis [3, 22, 29, 30] and feature extraction techniques [24, 33, 44]. However, the overall process remains largely unautomated, particularly the critical transplantation of code that implements useful functionality from a *donor* into a target system, which we call the *host*.

What if we could automate the process of extracting functionality from one system and transplanting it into another? This is the goal we set ourselves in this paper. That is, we are the first to develop and evaluate techniques to implement automated software transplantation from one system to another, which one of the authors proposed (hitherto unimplemented and unevaluated) in the keynote of the 2013 WCRE [28].

A programmer must first identify the entry point of code that implements a feature of interest. Given an entry point in the donor and a target implantation point in the host program, the goal of automated transplantation is to identify and extract an *organ*, all code associated with the feature of interest, then transform it to be compatible with the name space and context of its target site in the host. The programmer also supplies test suites that guide the search for donor code modifications required to make it fully executable (and pass all test cases) when deployed in the host.

This is a challenging vision of transplantation, because code from one system is unlikely to even compile when it is re-located into an unrelated foreign system without extensive modification, let alone execute and pass test cases. The extraction of the code also involves identifying all semantically required code and the successful insertion of the code organ into the host requires nontrivial modifications to the organ to ensure it adds the required feature without breaking existing functionality. In this paper, we present results that demonstrate that this vision of automated software transplantation is indeed achievable, efficient and potentially useful.

Feature identification is well studied [14, 16, 58]. Extracting a component of a system, given the identification of a suitable feature is also well studied, through work on slicing and dependence analysis [22, 24, 29, 33, 44]. The challenges of automatically extracting a feature from one system, transplanting it into an entirely different system, and usefully executing it in the organ beneficiary, however, have not previously been studied in the literature.

We developed μTrans, an algorithm for automated software transplantation, based on a new kind of genetic programming, augmented by a novel form of program slicing. μTrans synergizes analysis and testing: analysis extracts an 'organ', an executable slice from a donor; testing guides all phases of autotransplantation — identifying the organ in the donor, minimizing and placing the organ into an ice-box, and fi-

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771796

Automated Transplantation of Call Graph and Layout Features into Kate

Alexandru Marginean, Earl T. Barr, Mark Harman, and Yue Jia

UCL, Department of Computer Science, CREST Centre

Abstract. We report the automated transplantation of two features currently missing from Kate: call graph generation and automatic layout for C programs, which have been requested by users on the Kate development forum. Our approach uses a lightweight annotation system with Search Based techniques augmented by static analysis for automated transplantation. The results are promising: on average, our tool requires 101 minutes of standard desktop machine time to transplant the call graph feature, and 31 minutes to transplant the layout feature. We repeated each experiment 20 times and validated the resulting transplants using unit, regression and acceptance test suites. In 34 of 40 experiments conducted our search-based autotransplantation tool, μScalpel, was able to successfully transplant the new functionality, passing all tests.

1 Introduction

We recently introduced a search based technique for automated software transplantation [2,7]. Guided by dependence analysis and testing, our approach uses a variant of genetic programming to identify and extract useful functionality from a donor program, and transplant it into a (possibly unrelated) host program. We implemented our approach as a tool called μScalpel, which is publicly available [1].

In this challenge paper, we illustrate the way in which realistic, scalable, and useful real-world transplantation can be achieved using μScalpel. We apply our tool to the SSBSE 2015 Challenge program `Kate`[1], a popular text editor based on KDE. Its rich feature set and available plugins make it a popular, lightweight IDE for C developers. We perform two automated transplantations using μScalpel. In the first one, we transplant call graph drawing ability from the GNU utility program `cflow`, to augment `Kate` with the ability to construct and display call graphs.

This is a useful feature for a lightweight IDE, like `Kate`, and would clearly be nontrivial to implement from scratch. Using our search based autotransplantation, μScalpel, the developer merely needs to identify the entry point of the source code in the donor program (`cflow` in this case) and the tool will do the rest; extracting the relevant code, matching names spaces between host and donor and executing regressions, unit and acceptance tests. Like much previous work on genetic programming [12], our approach relies critically on the availability of high quality test suites. We do not directly address this issue in the present paper, but believe

[1] http://kate-editor.org

Automated Software Transplantation

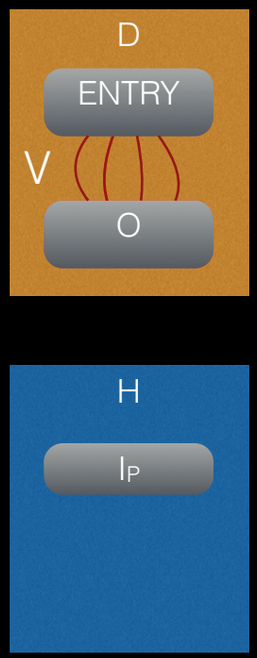
Donor

Host

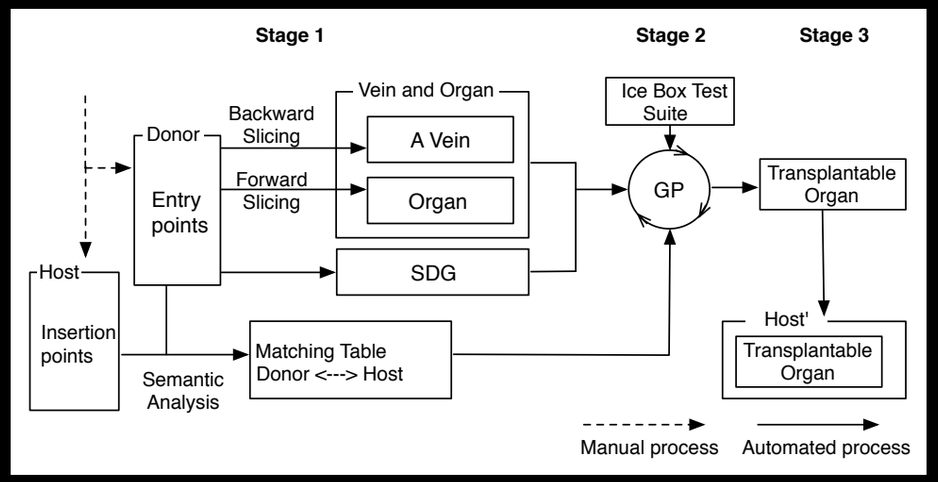
Organ

Vein

I_P



Approach



Validation

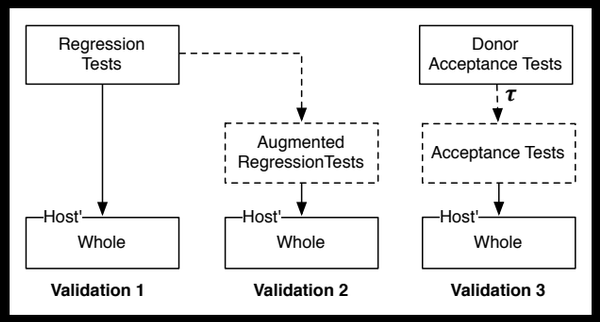
4 validation steps:

Regression Tests;

Augmented Regression Tests;

Acceptance Tests;

Manual Validation;



Results

Case Studies

		All	Test Suites			Time
Donor	Host	Passed	Regression	Regression++	Acceptance	Avarage
x264	VLC	1	1	1	1	26 (hours)
cflow	Kate	16	20*	17	18	101
Indent	Kate	18	20*	18	19	31