DSI: Automated Detection of Dynamic Data Structures in C Programs and Binary Code

Thomas Rupprecht, Jan H. Boockmann, David H. White, and **Gerald Lüttgen**

Software Technologies Research Group
University of Bamberg, Germany
https://www.swt-bamberg.de

Shonan Meeting 100, 2nd October 2017

Research support provided by the German Science Foundation (DFG grant no. LU 1748/4-1)

tomorrow's software technologies

Motivation

Goal:

- Identifying dynamic data structures in pointer programs, especially for C programs and x86 binaries

Envisaged applications, targeting user-/kernel-space software:

- Heap visualization
- Program comprehension
- Reverse engineering
- Program verification

Challenges:

- **Fine-grained heaps** due to, e.g., pointer arithmetic, type casts, and customized memory allocators
- **Degenerate shapes** during data structure operations

Related Work: Dynamic Analyses on Memory Snapshots

ARTISTE (Caballero et al., 2012), MemPick (Haller et al., 2016):

- Analyze x86 binaries for reverse engineering and security applications
- Do *not* deal with fine-grained heaps (see, e.g., the Linux CDLL)
- Either address degenerate shapes (MemPick, detection of quiescent periods) or avoid them (ARTISTE, but does *not* address nesting)

Heapviz (Aftandilian et al., 2010), HeapDbg (Marron et al., 2013):

- Visualize entire runtime heaps of Java byte code and, resp., C#
- Fold points-to graphs and layout them based on graph dominators
- Do *not* deal with fine-grained heaps
- Infer shapes superficially (HeapDbg, works not unlike MemPick)

DSI – Data Structure Investigator

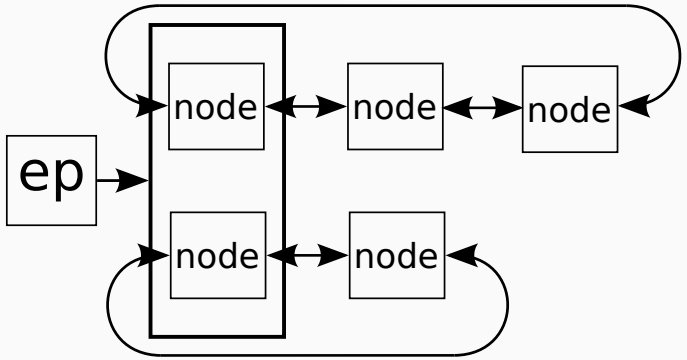
A dynamic analysis inspired by 'heap abstraction' and 'pattern matching'

Scope:

- Singly- and doubly-linked lists (cyclic, w/ head and/or tail pointers)
- Skip lists and (binary) trees
- Combinations of such data structures via nesting (indirect, overlay)

Contributions:

- Memory abstraction, **strand graphs**, to handle fine-grained heaps
  - Lifts the assumption that each memory chunk contains a single node
  - Tracks **cells**, i.e., individual structs, instead of whole nodes



DSI – Data Structure Investigator

A dynamic analysis inspired by 'heap abstraction' and 'pattern matching'

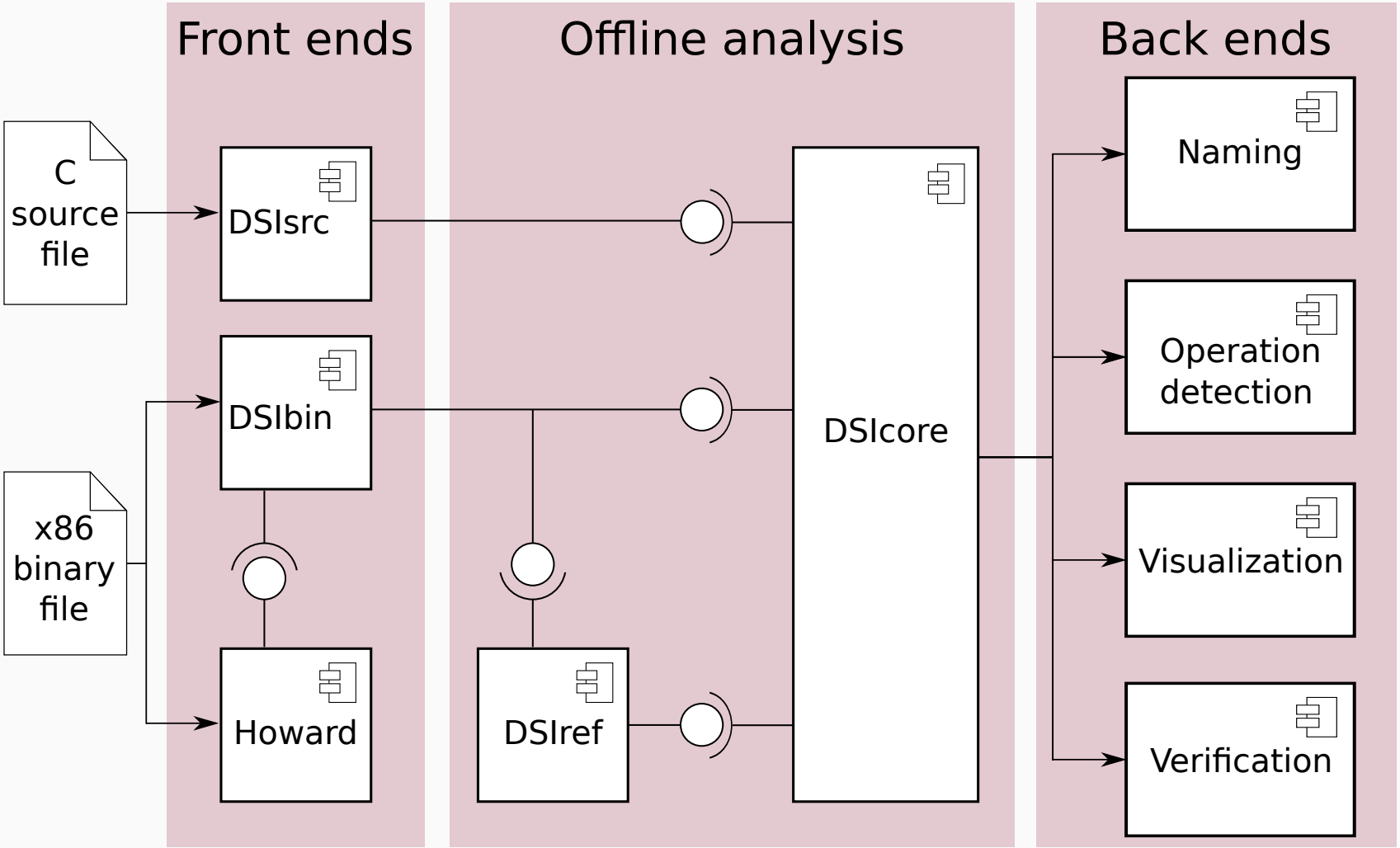
Scope:

- Singly- and doubly-linked lists (cyclic, w/ head and/or tail pointers)
- Skip lists and (binary) trees
- Combinations of such data structures via nesting (indirect, overlay)

Contributions:

- Memory abstraction, **strand graphs**, to handle fine-grained heaps
  - Lifts the assumption that each memory chunk contains a single node
  - Tracks **cells**, i.e., individual structs, instead of whole nodes
- Novel **evidence-based approach** that is robust wrt. degenerate shapes
  - Discovering evidence for each kind of data structure
  - Reinforcing evidence by structural and temporal repetition

The DSI Tool Suite



DSIcore

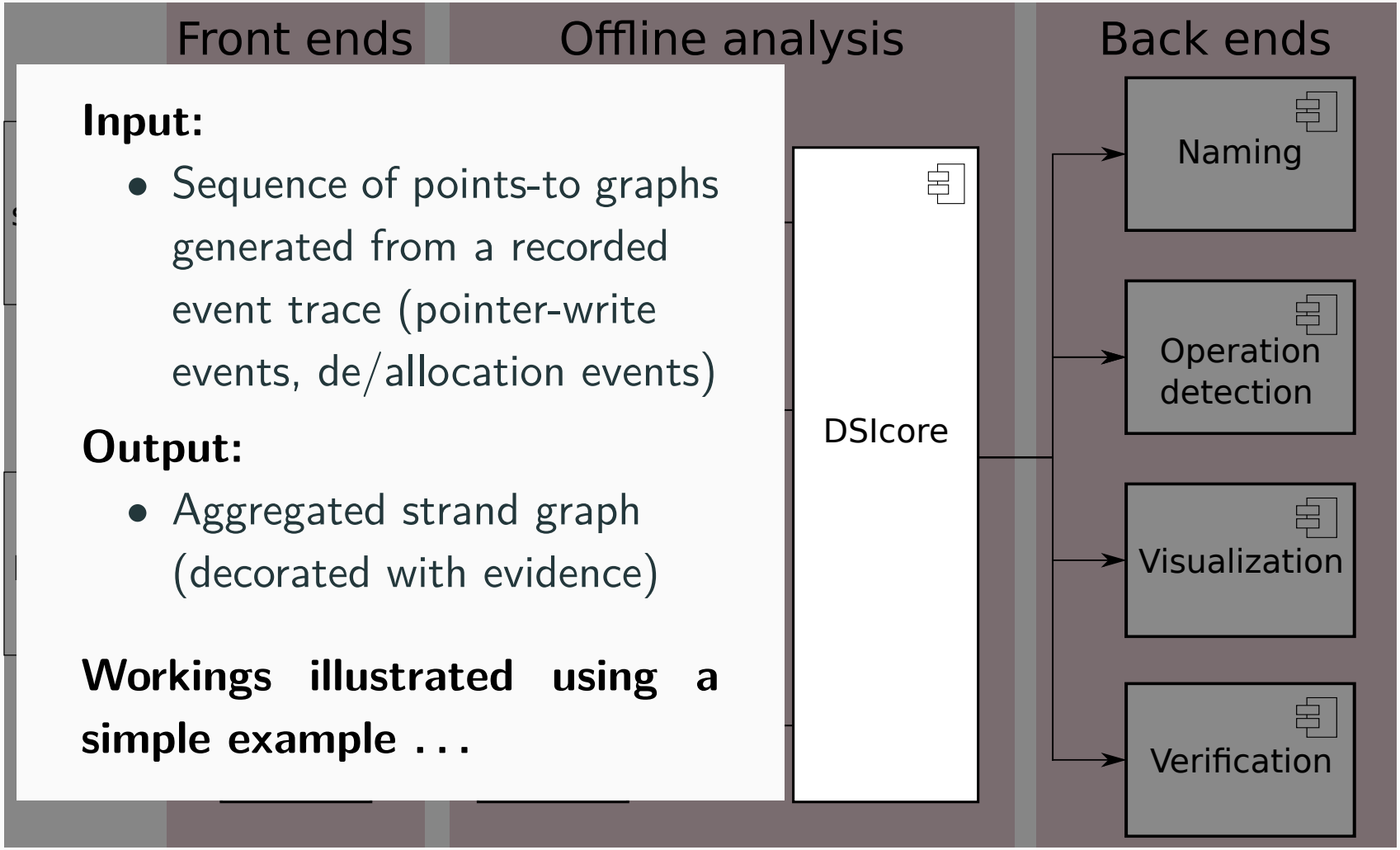
Input:

- Sequence of points-to graphs generated from a recorded event trace (pointer-write events, de/allocation events)

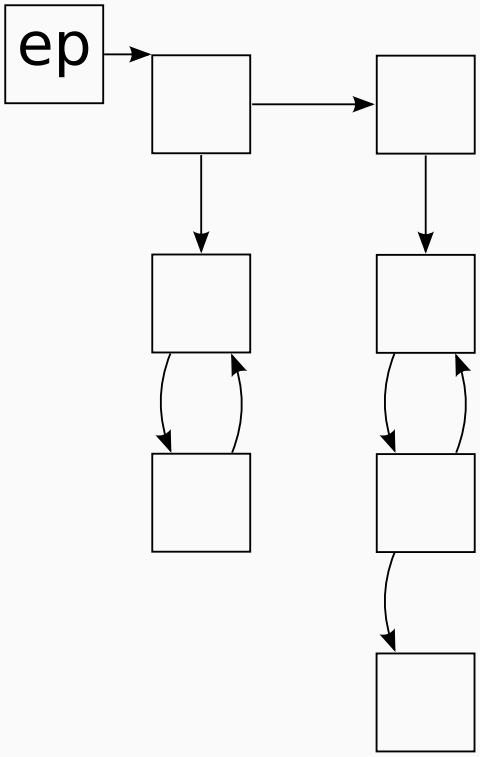
Output:

- Aggregated strand graph (decorated with evidence)

Workings illustrated using a simple example . . .



Illustrative Example

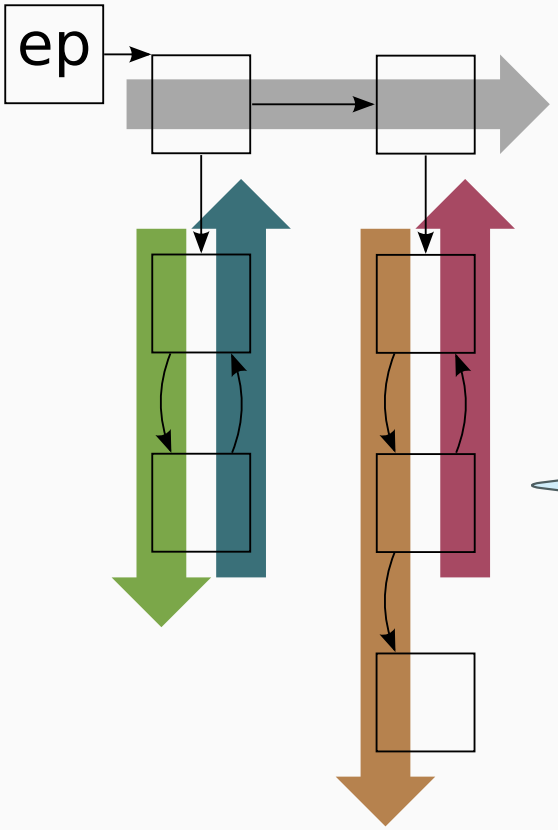


Points-to Graph

Points-to graph – abstraction of program memory:

- **Memory chunks** (vertices)
- **Pointers** (edges)
- **Entry points** (ep)

Illustrative Example

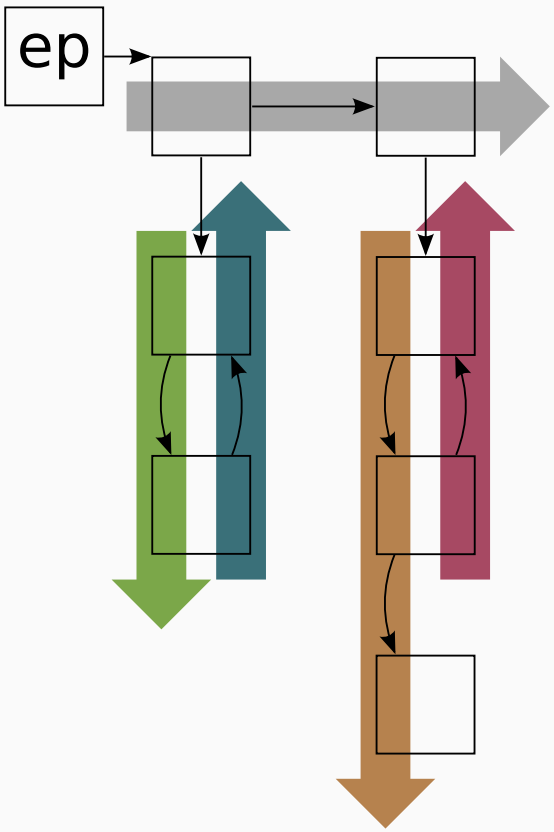


Points-to Graph

Strands – atomic building blocks

- Essentially SLLs

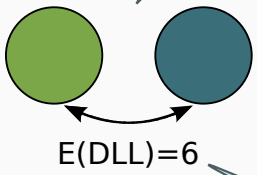
Illustrative Example



Points-to Graph

Strand connection

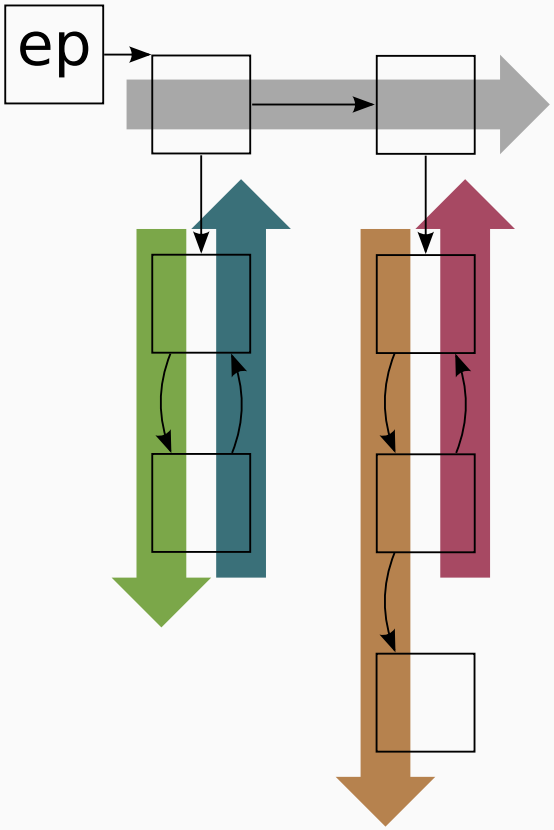
- Relationship between a pair of strands
- Count evidence for each data structure type



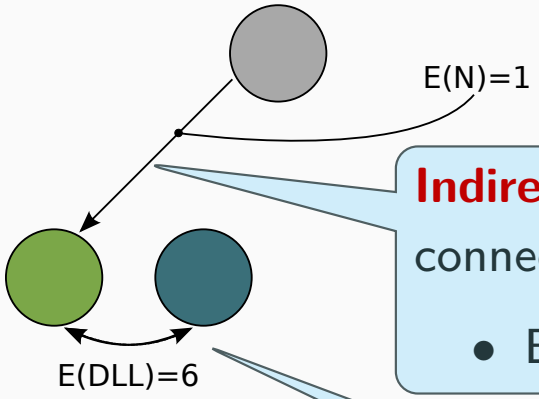
Overlay (bi-directional) connection

- E.g., DLL
- Why evidence = 6?

Illustrative Example



Points-to Graph



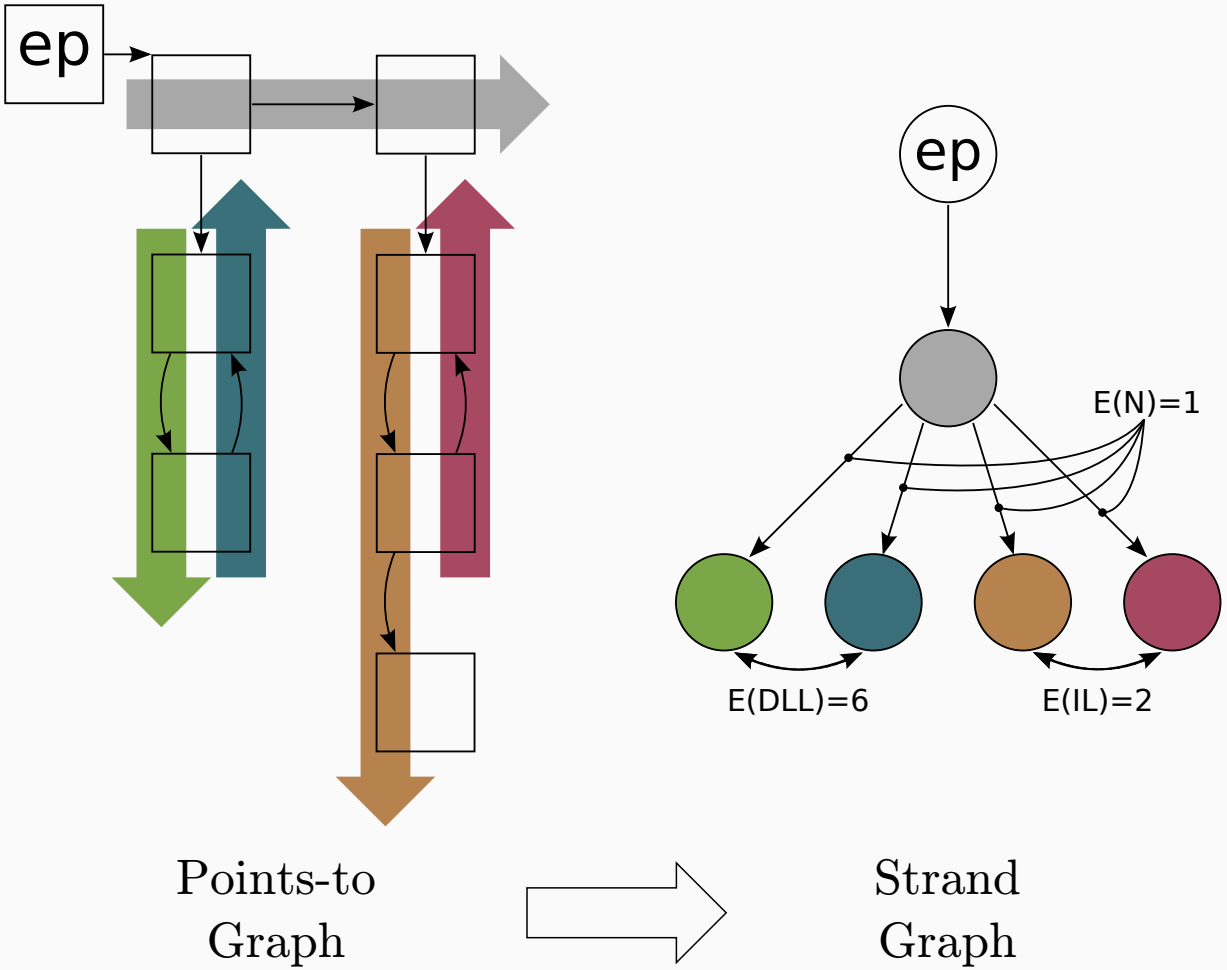
Indirect (uni-directional) connection

- E.g., nesting (N)

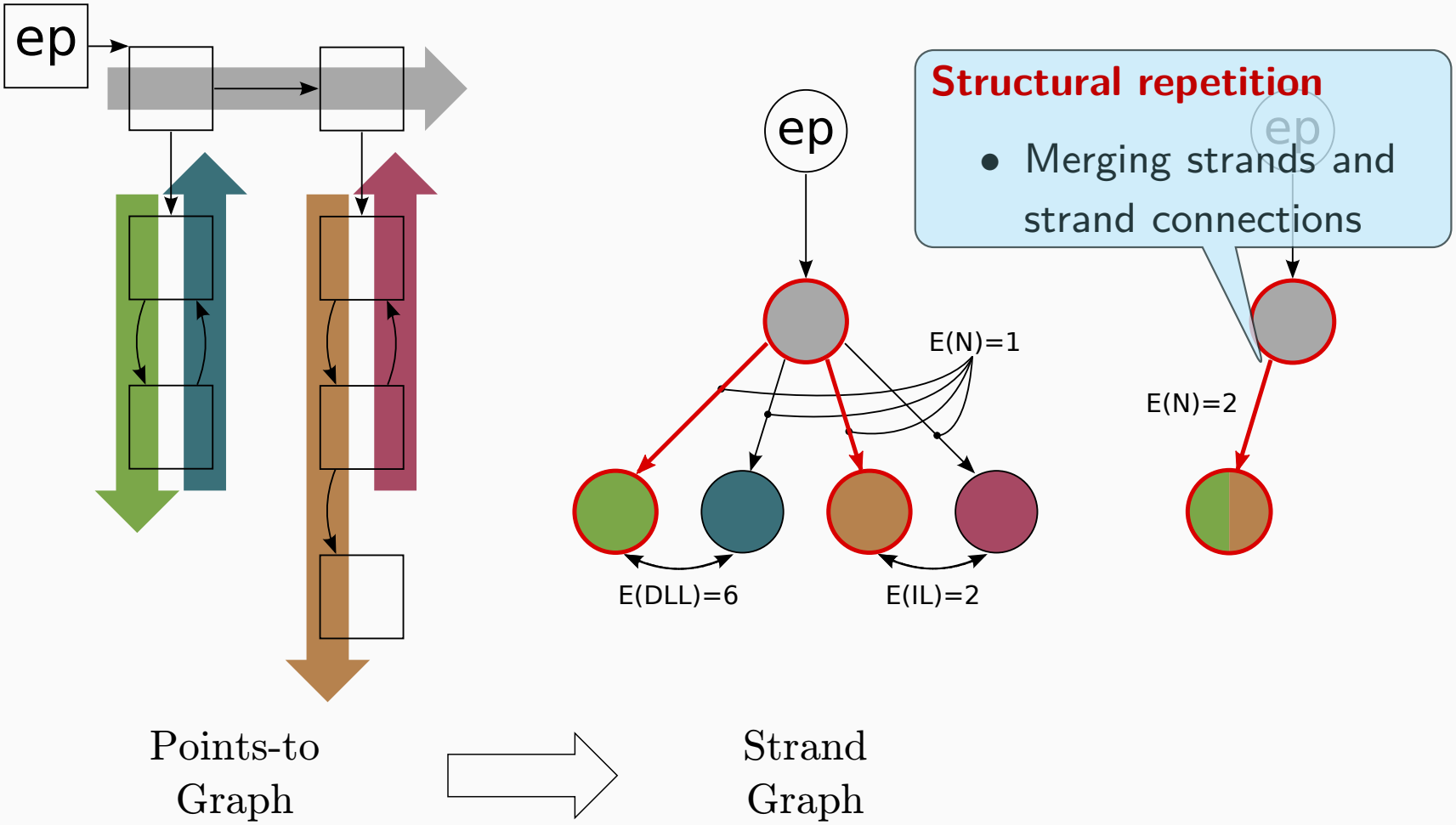
Overlay (bi-directional) connection

- E.g., DLL

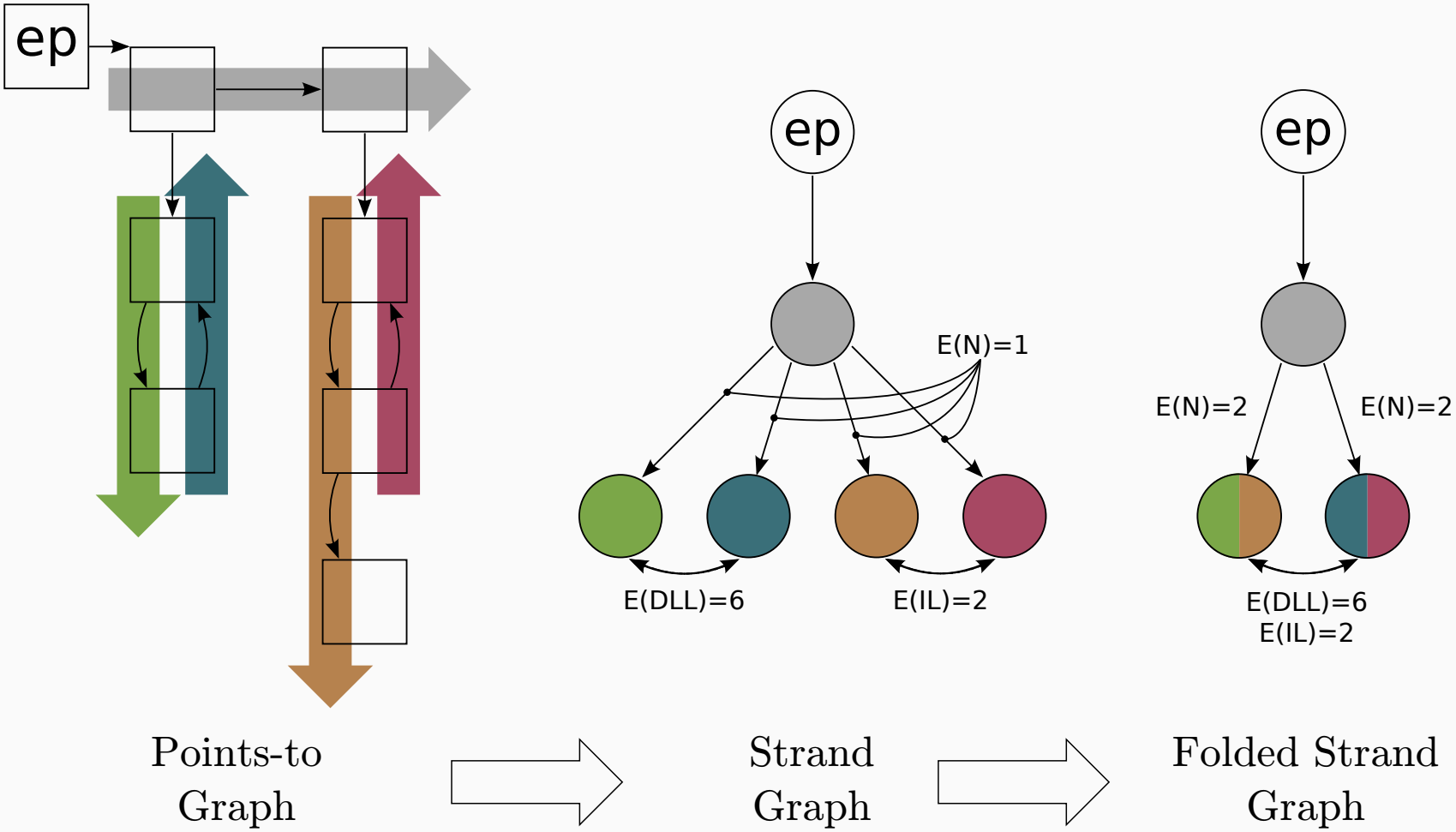
Illustrative Example



Illustrative Example

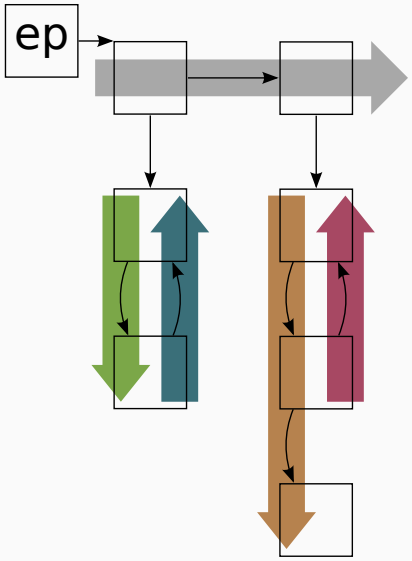


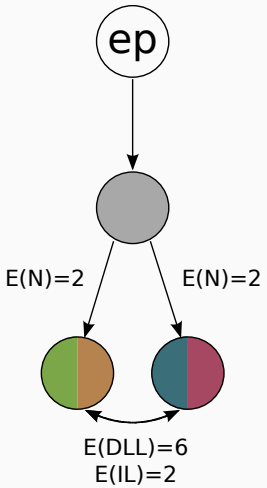
Illustrative Example



Illustrative Example

Time t

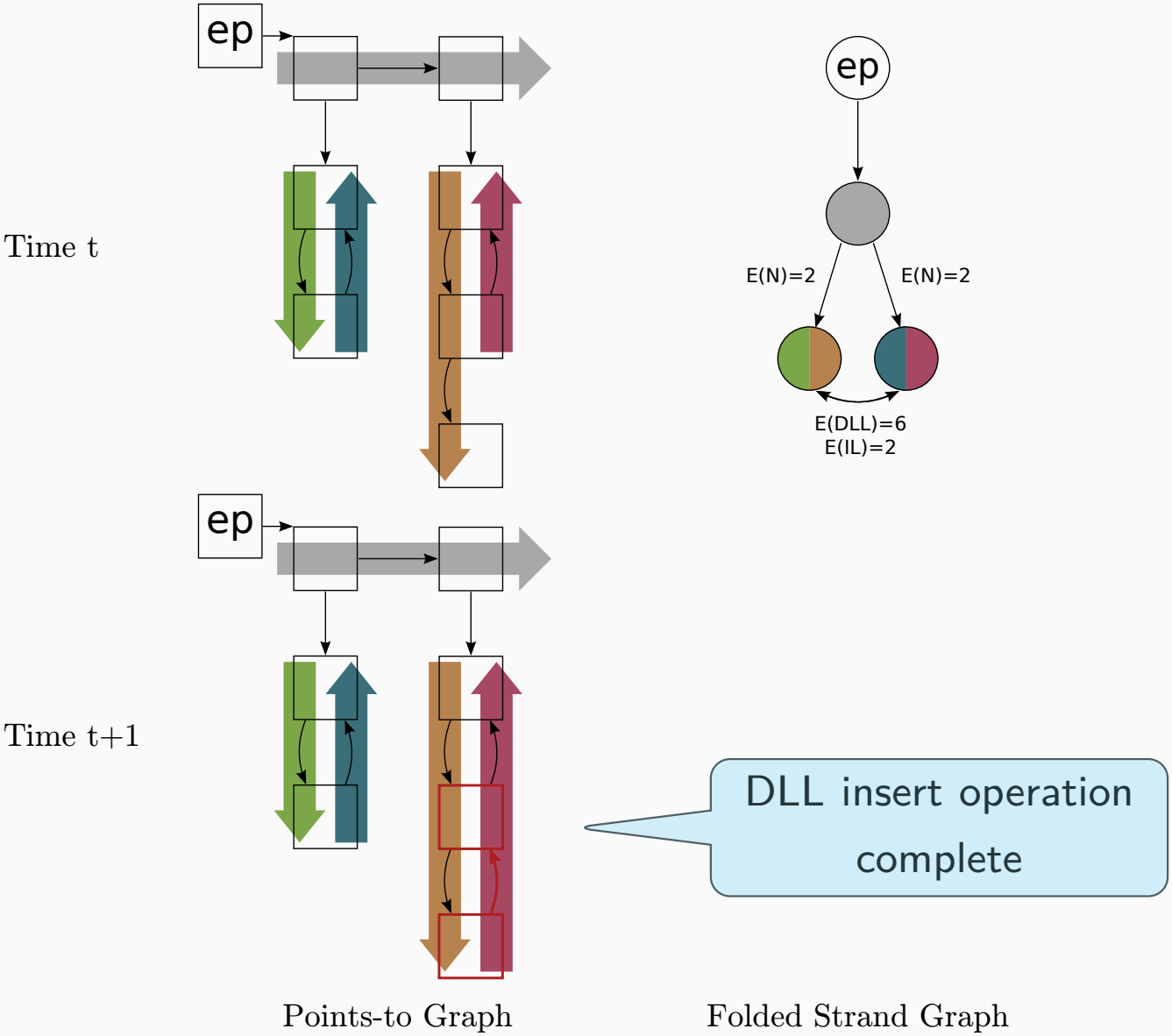




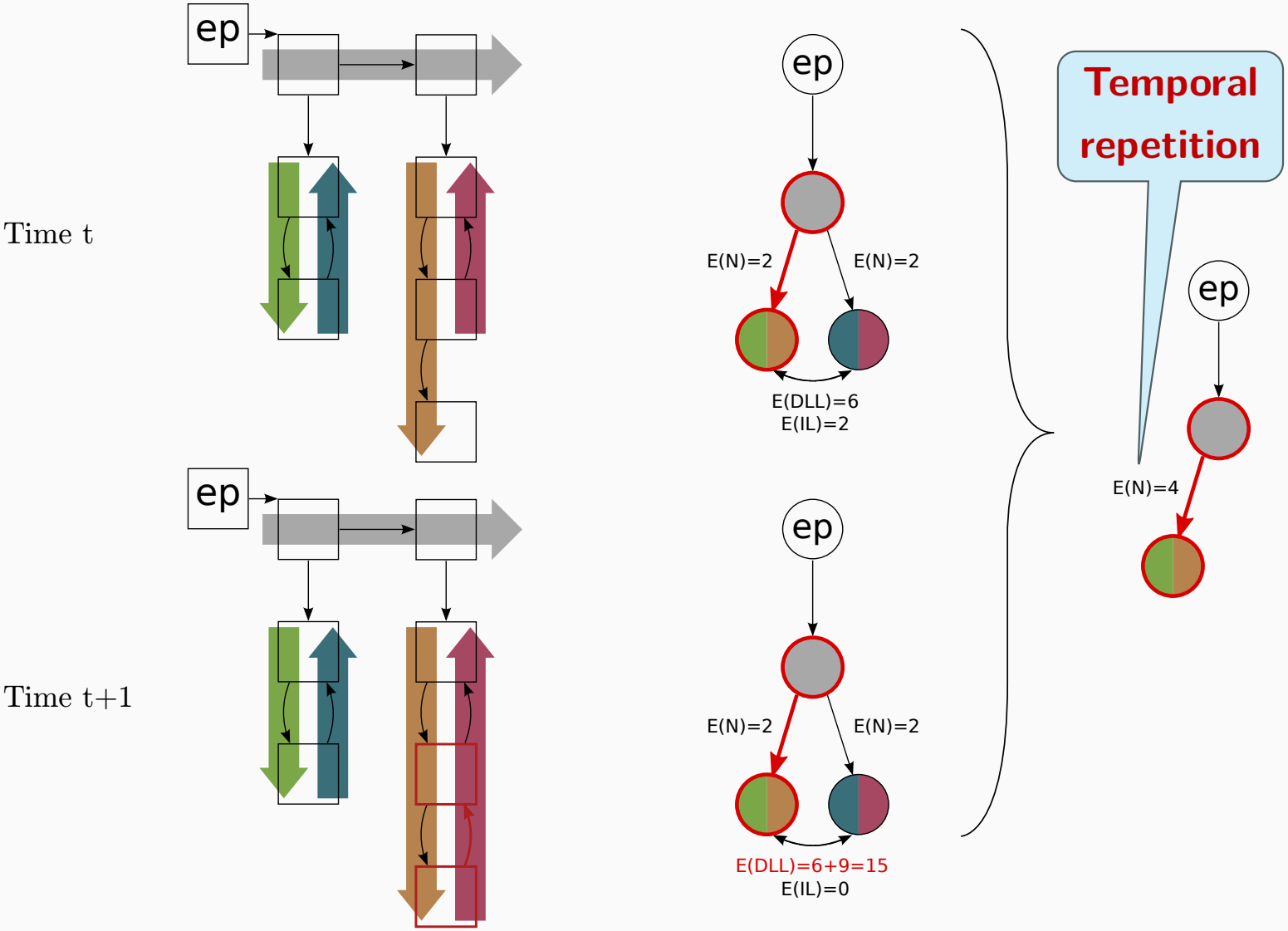
Points-to Graph

Folded Strand Graph

Illustrative Example



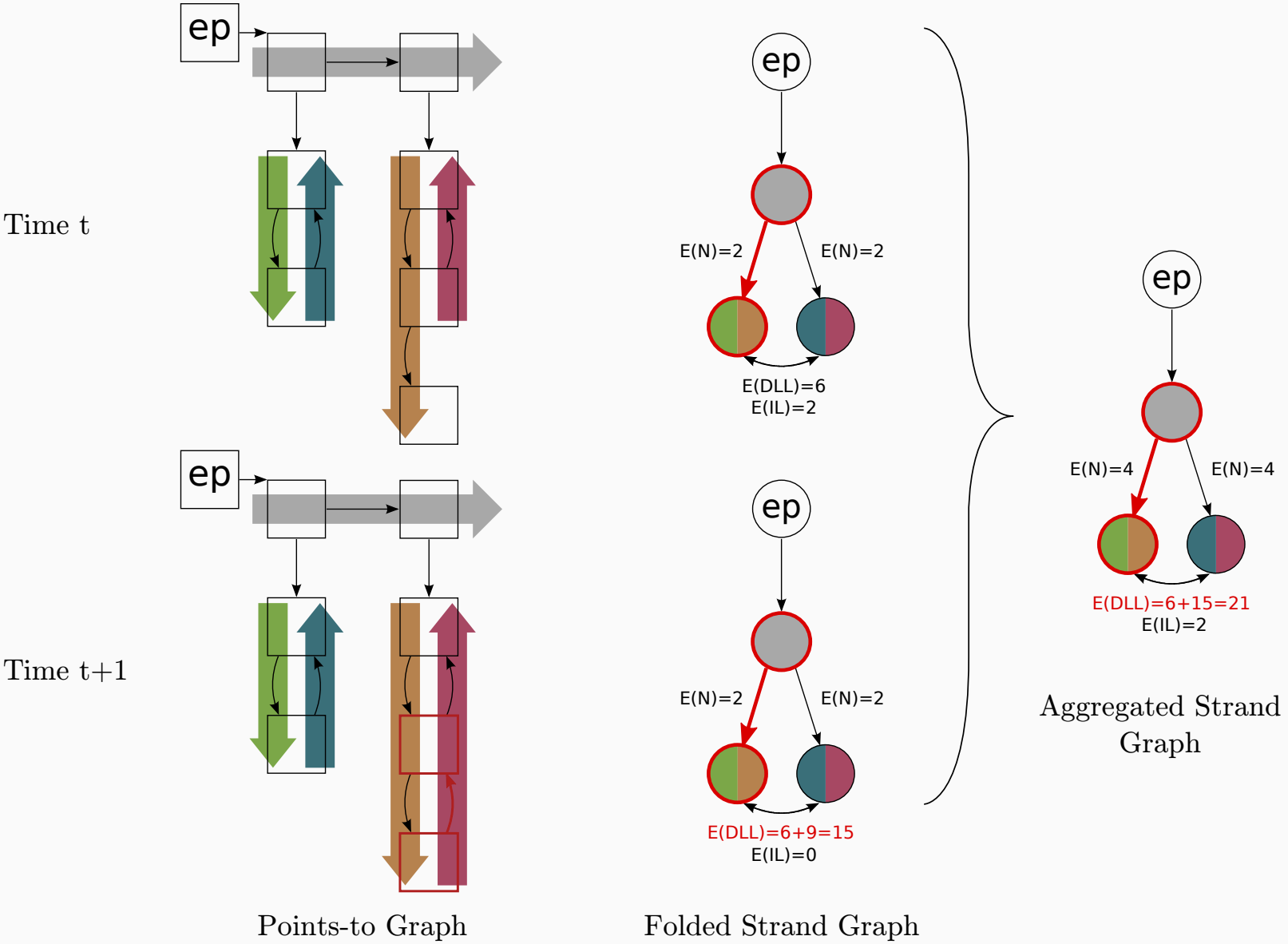
Illustrative Example



Points-to Graph

Folded Strand Graph

Illustrative Example

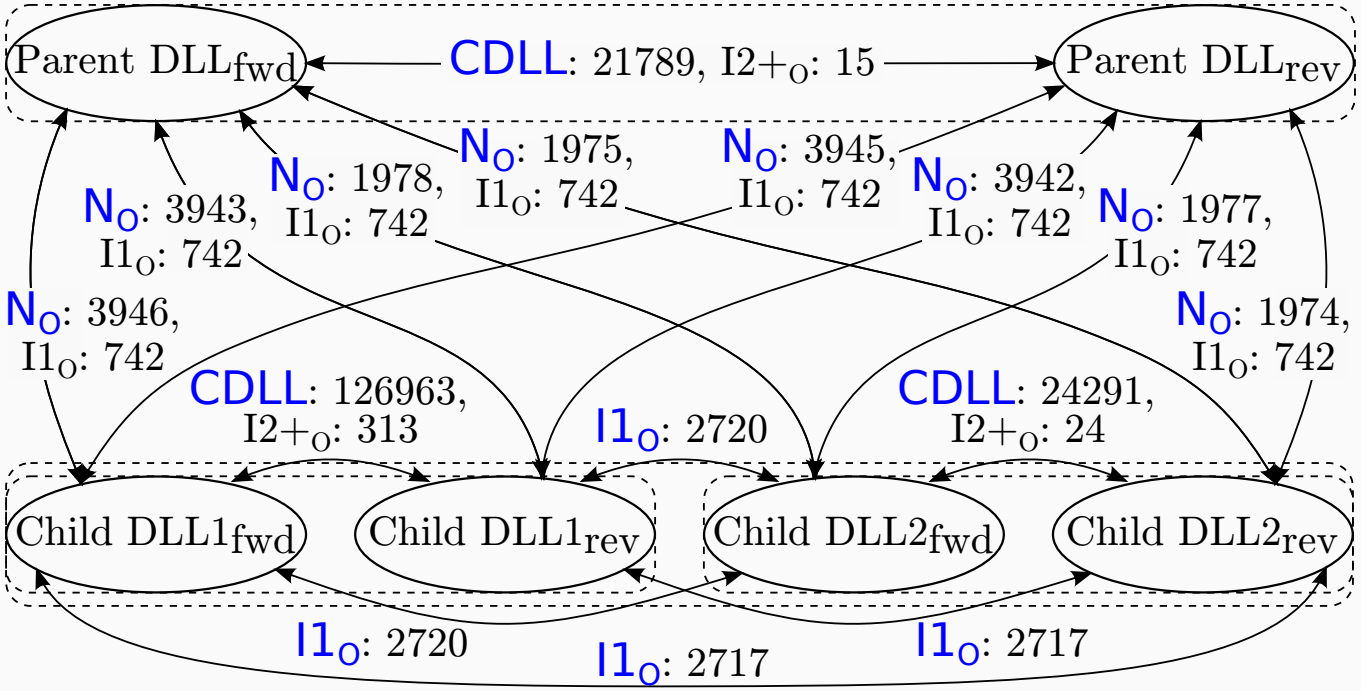


Example (libusb)

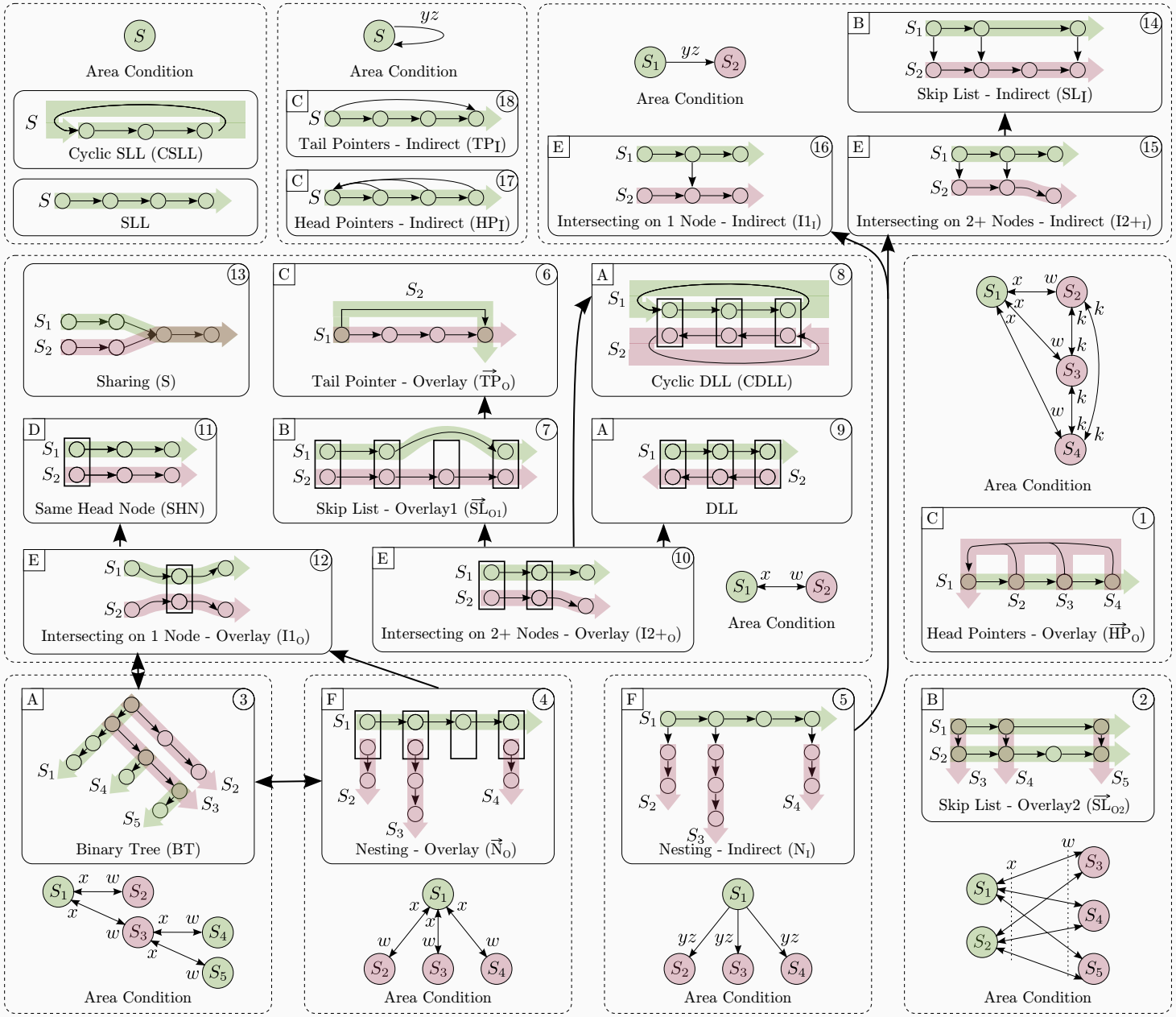
Usb device library providing access from user-space (7K LOC C):

- Uses a parent CDDL with two child CDLLs (one for devices, one for file descriptors)
- Employs Linux CDDLs that are embedded in surrounding structs, i.e., our cell-based memory abstraction is required

Aggregated strand graph:



Taxonomy & Matching of Data Structures



Taxonomy & Matching of Data Structures

$\text{DLL.}\textsc{areaCondition}(C_{\text{init}}, \_, \_, \_) =$

$\quad$ **if** $C_{\text{init}} = S_1 \overset{xw}{\longleftrightarrow} S_2$ **then ret** $\{C_{\text{init}}\}$ **else ret** $\emptyset$

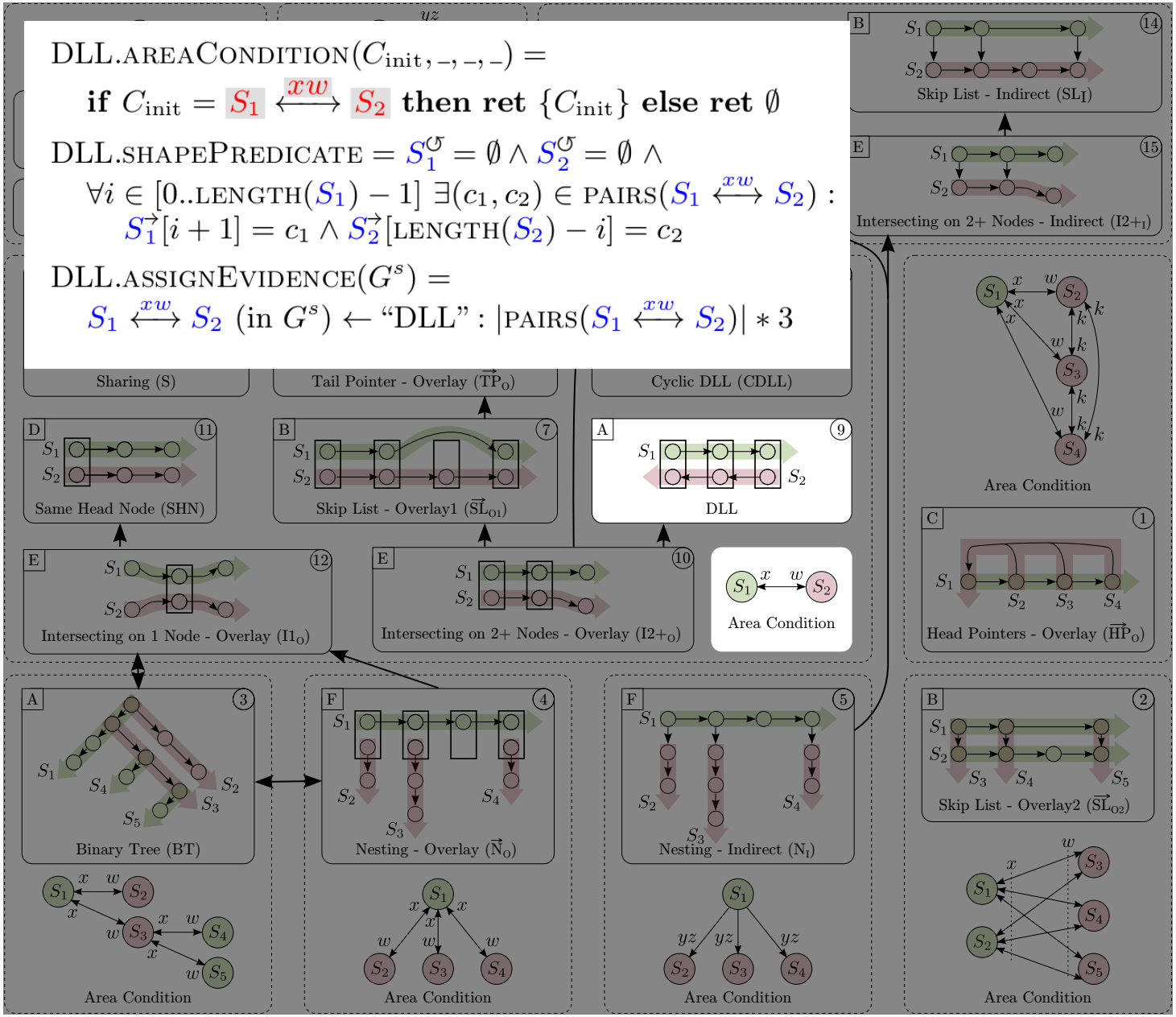
$\text{DLL.}\textsc{shapePredicate} = S_1^{\circlearrowleft} = \emptyset \wedge S_2^{\circlearrowleft} = \emptyset \wedge$

$\quad \forall i \in [0..\textsc{length}(S_1) - 1]\ \exists (c_1, c_2) \in \textsc{pairs}(S_1 \overset{xw}{\longleftrightarrow} S_2) :$

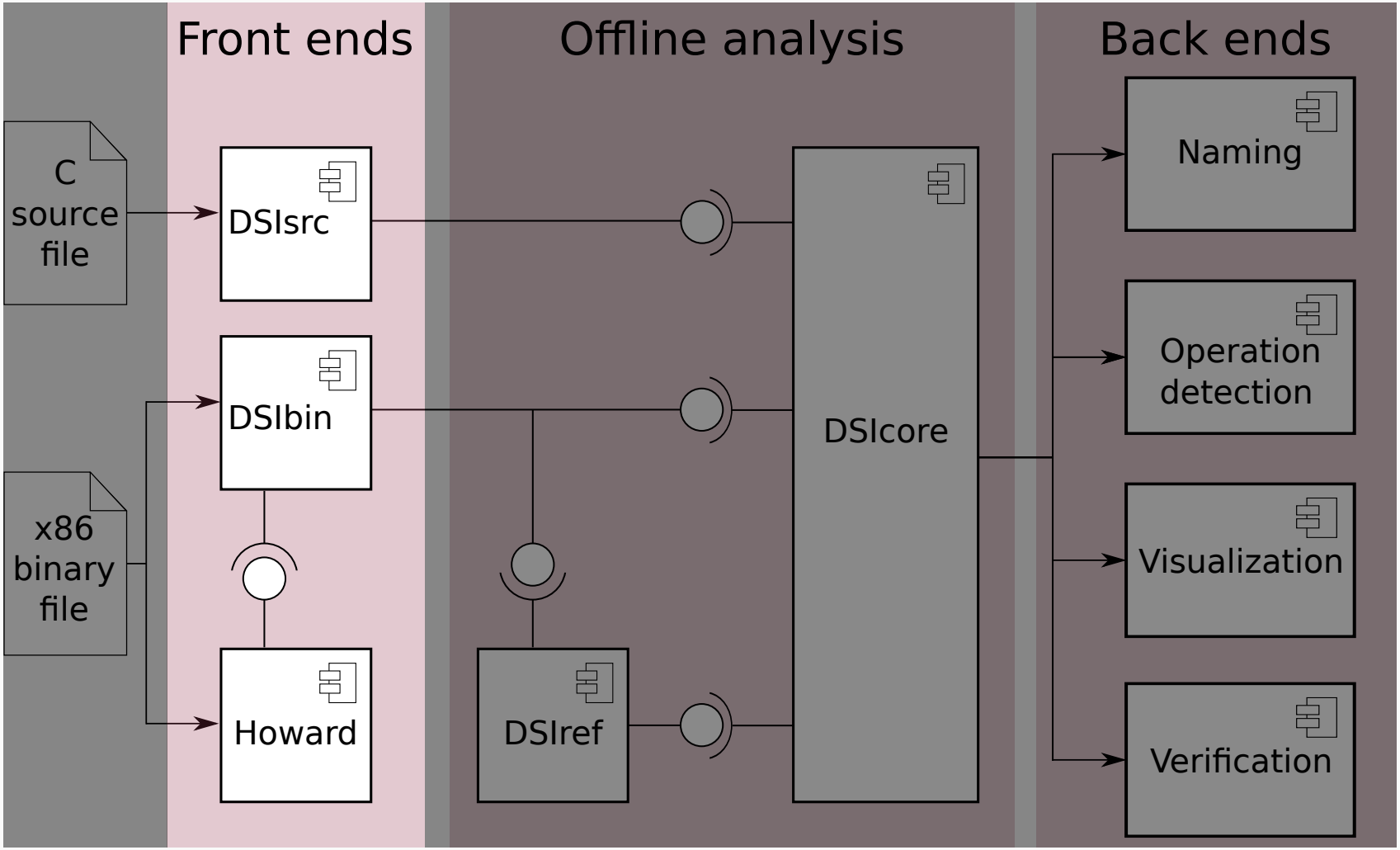
$\quad\quad \vec{S_1}[i+1] = c_1 \wedge \vec{S_2}[\textsc{length}(S_2) - i] = c_2$

$\text{DLL.}\textsc{assignEvidence}(G^s) =$

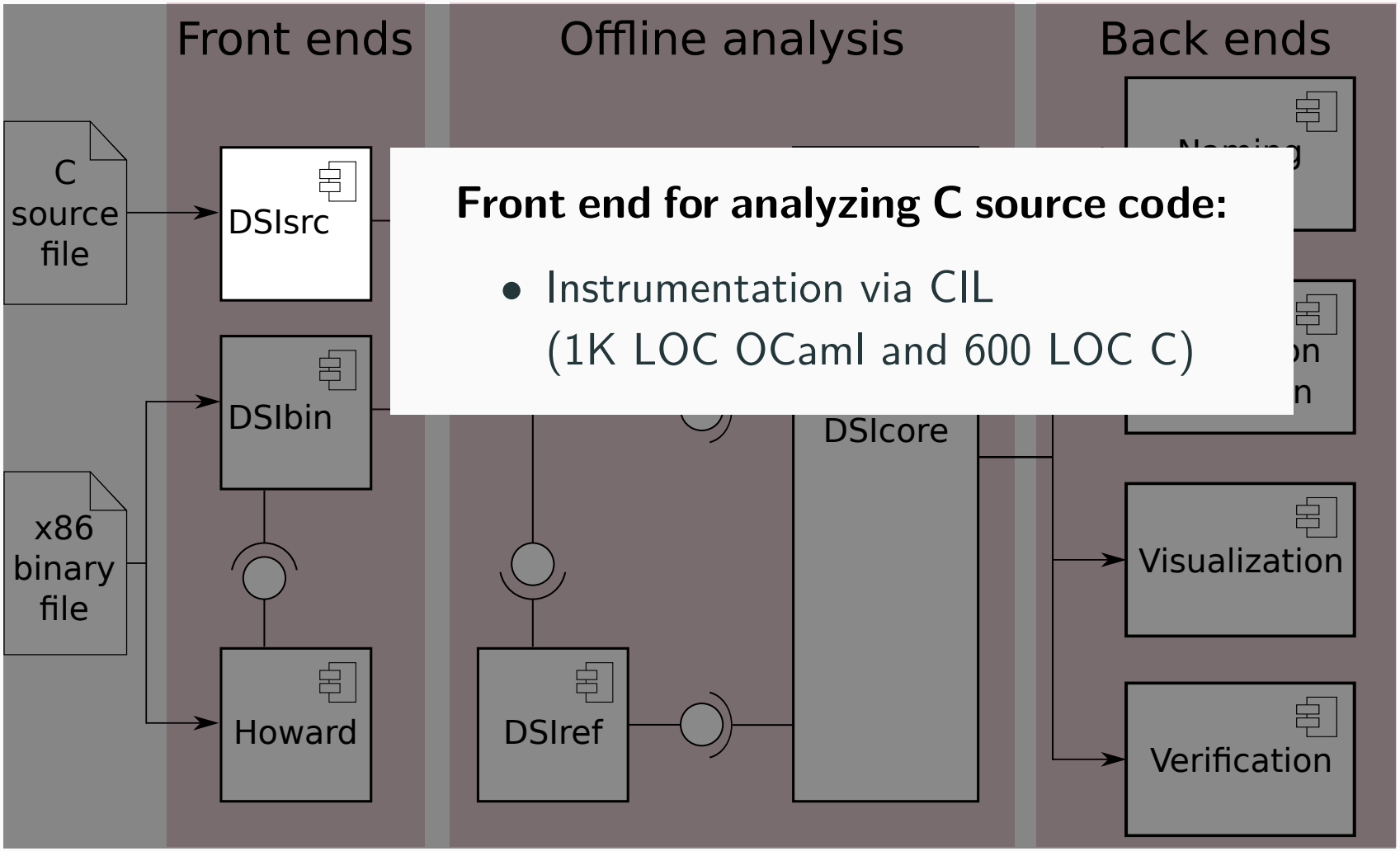
$\quad S_1 \overset{xw}{\longleftrightarrow} S_2$ (in G^s) $\leftarrow$ "DLL" : $|\textsc{pairs}(S_1 \overset{xw}{\longleftrightarrow} S_2)| * 3$



DSI Front Ends



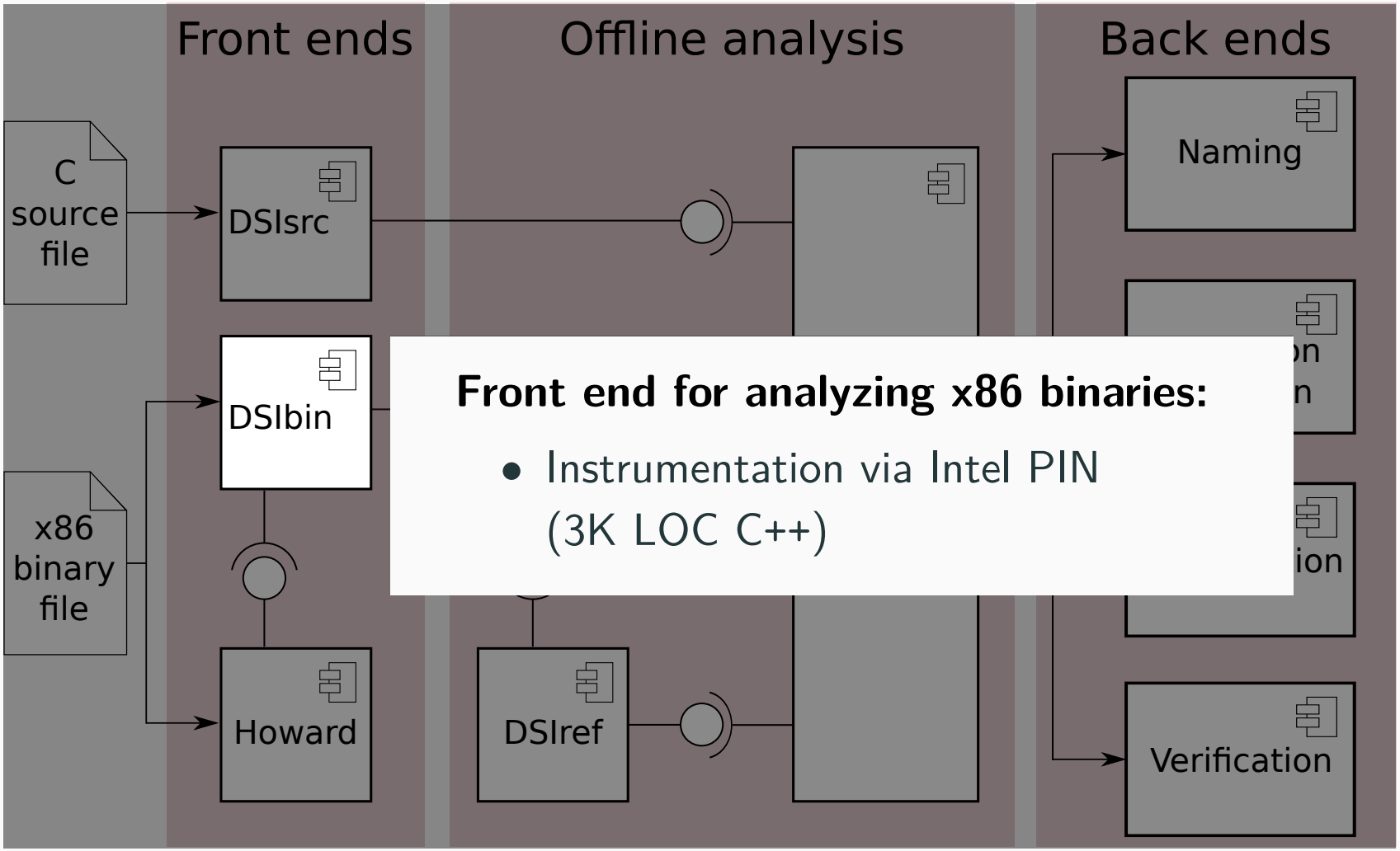
DSI Front Ends



Front end for analyzing C source code:

- Instrumentation via CIL (1K LOC OCaml and 600 LOC C)

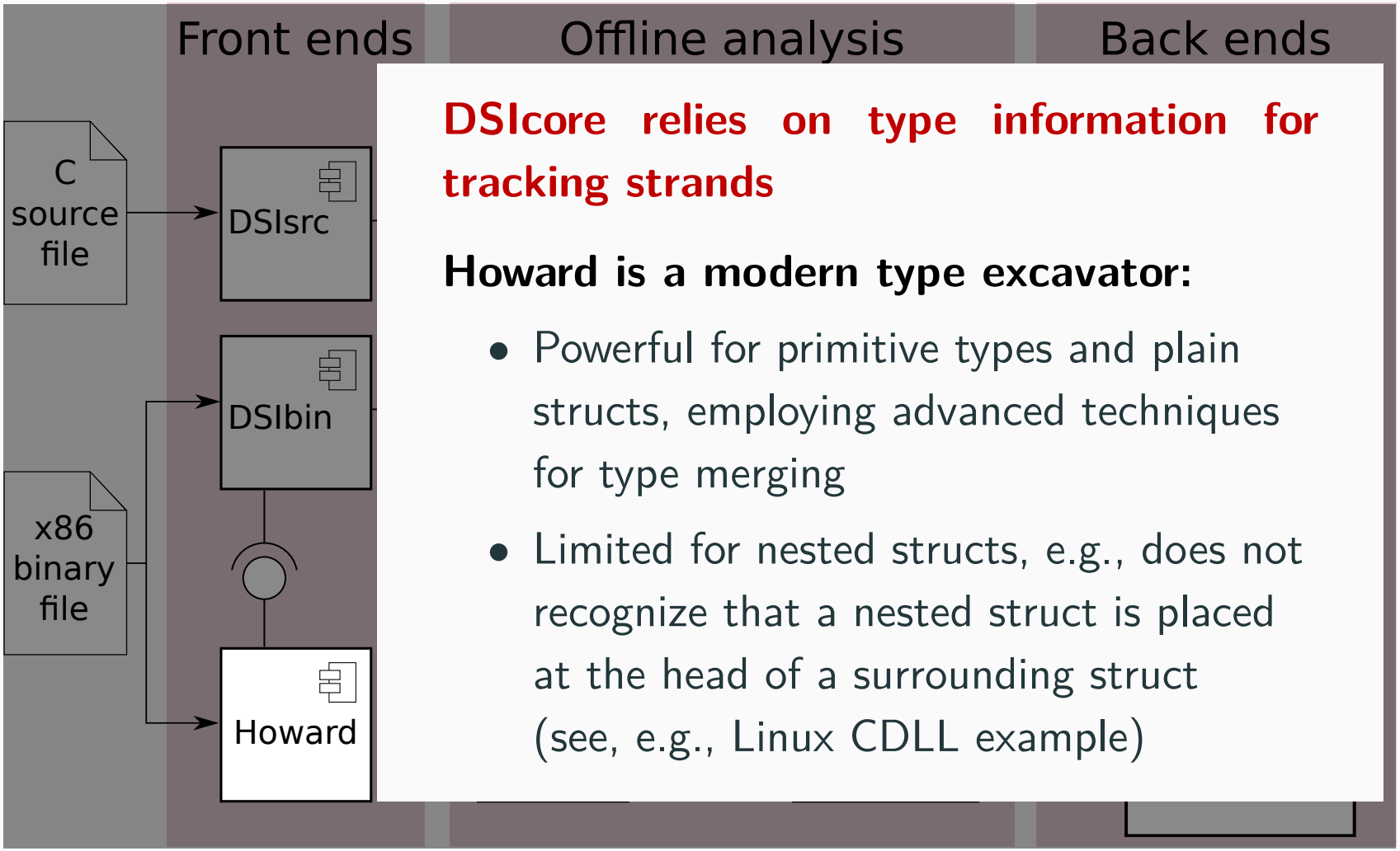
DSI Front Ends



Front end for analyzing x86 binaries:

- Instrumentation via Intel PIN (3K LOC C++)

DSIbin: Recovering Type Information



DSIcore relies on type information for tracking strands

Howard is a modern type excavator:

- Powerful for primitive types and plain structs, employing advanced techniques for type merging
- Limited for nested structs, e.g., does not recognize that a nested struct is placed at the head of a surrounding struct (see, e.g., Linux CDLL example)

DSIref: Refining Howard's Type Information

Novel method for type refinement, which obtains high-level data structure information from DSIcore in order to infer low-level type information

Input: Incomplete type information inferred by Howard

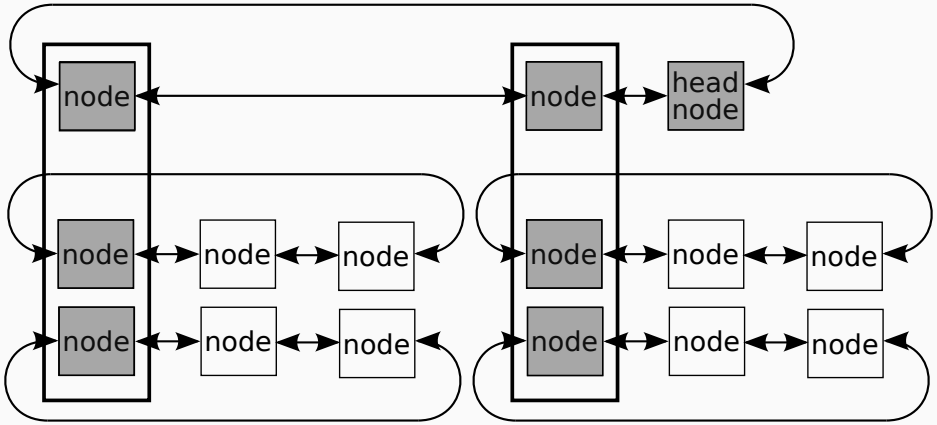
Output: Refined/completed type information

Workings in a nutshell:

- Make educated guesses on the missing types within structs (resulting in type hypotheses)
- Evaluate the (many) hypotheses with DSIcore wrt. their plausibility
- Choose the structurally most complex data structure

Example (from Predator/Forester git repository)

CDLL parent with two CDLL children (same as libusb):

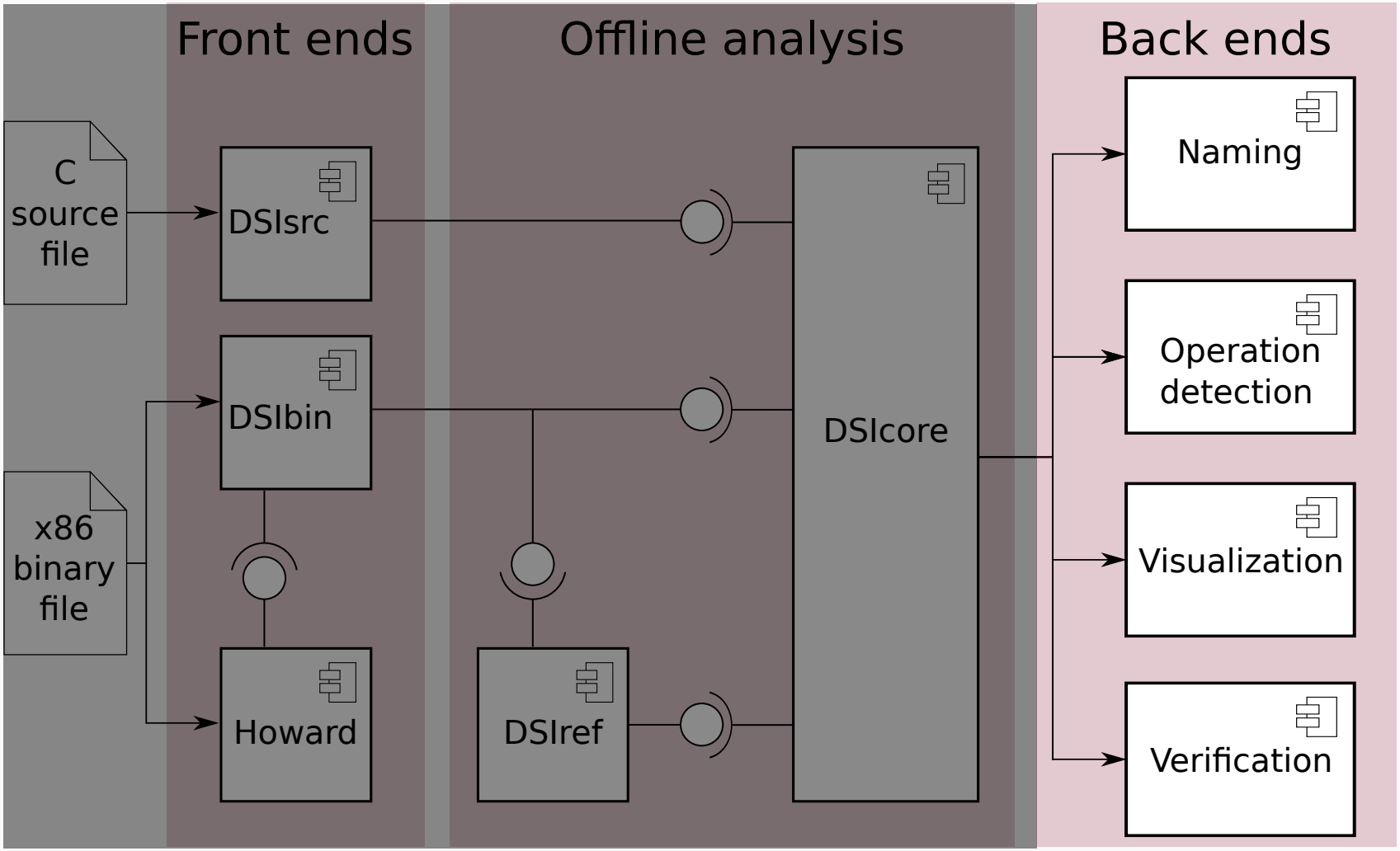


(white nodes are typed by Howard, grey nodes are *only* typed by DSIref)

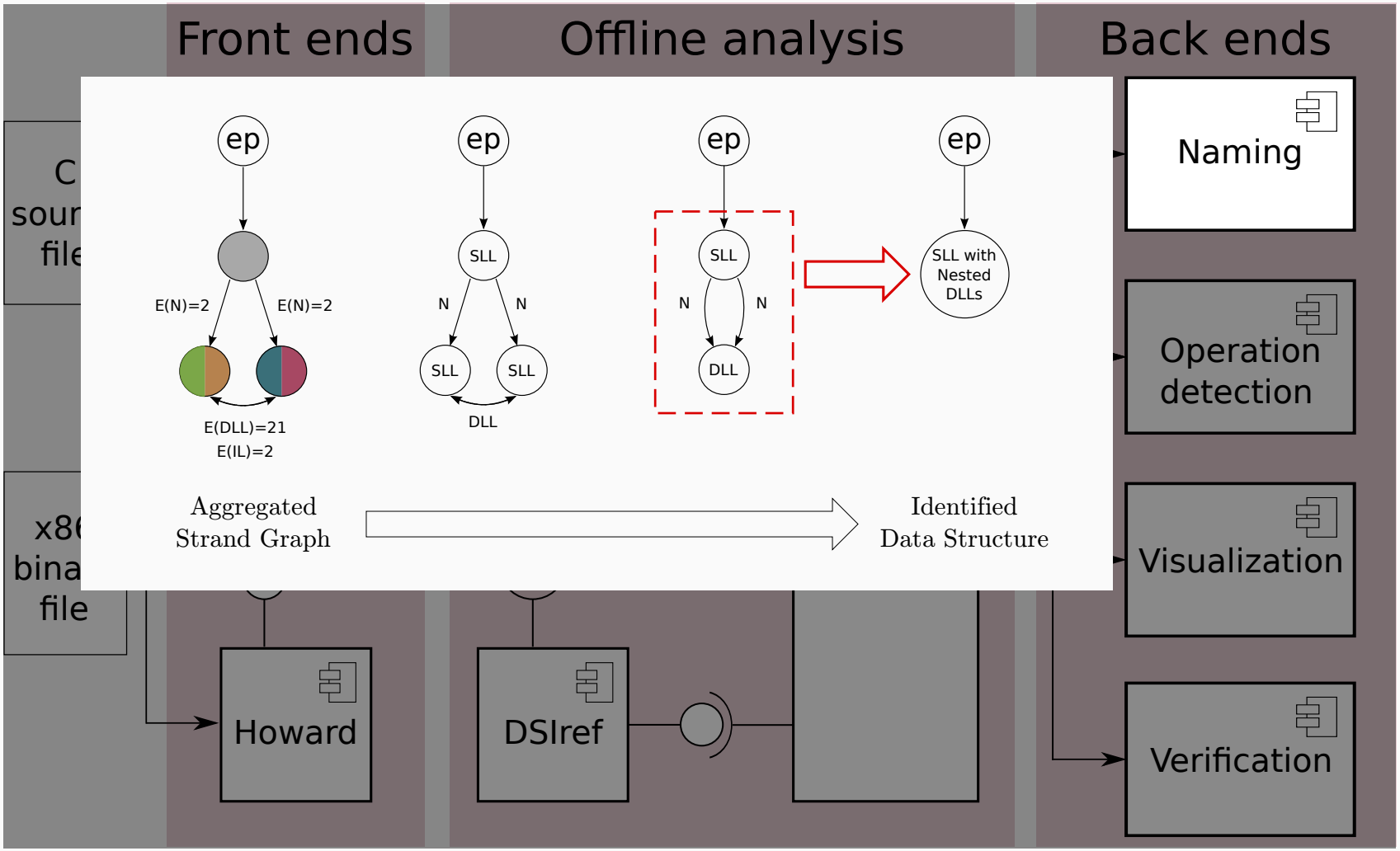
Howard does not detect the parent nested structs and the external head struct to be of same type, thus:

- Cyclicity property of the DLLs is missed
- Overlay nesting is missed ('nesting on indirection' reported instead)

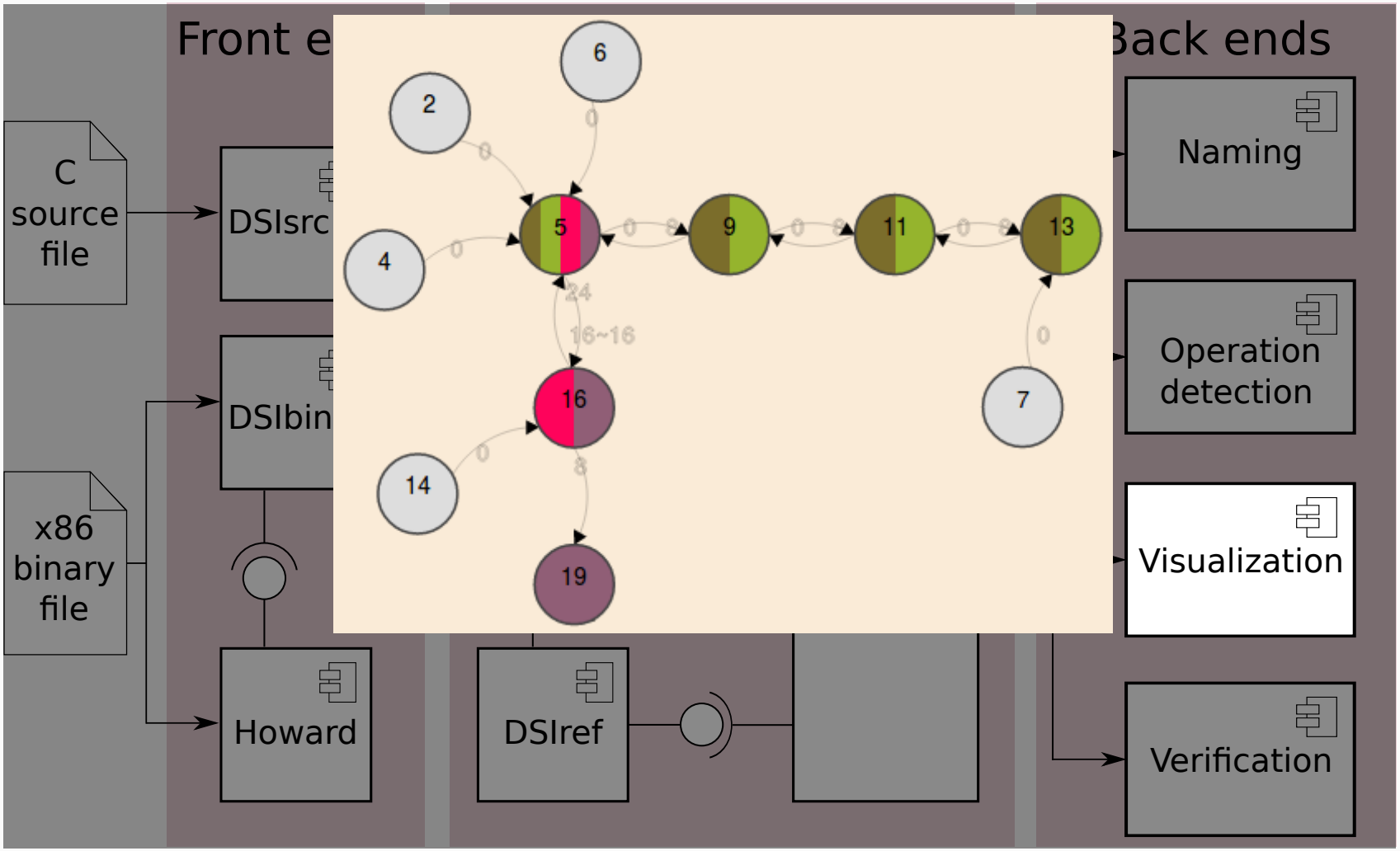
DSI Back Ends



Naming Back End



Visualization Back End



Conclusions

DSI is a novel dynamic analysis for identifying dynamic data structures in pointer programs:

- Introduces **strand abstraction** and **evidence-based identification** (using a sophisticated taxonomy)
- Robustly identifies data structures despite degenerate shapes (shown via extensive benchmarking)
- Copes with the 'dirtiness' / 'efficiencies' of (low-level) C code
- Is applicable to x86 binaries, thanks to advanced type refinement
- Requires work on speeding up the analysis, e.g., via parallelization

DSI has potential for a broad range of applications:

- Heap visualization, program comprehension
- Reverse engineering, malware identification
- Program verification

Future Work

- Improve the runtime of DSI, e.g., via parallelization
- Investigate relation to static shape analysis tools such as Predator (Dudka et al., 2013) and Forester (Holík et al., 2013), which focus on program verification rather than program comprehension

Three-year duration follow-up DFG project, granted last month:

- Cover richer data structures, e.g., hash maps (arrays), sorted lists, and balanced trees
- Integrate DSI with IDA Pro (disassembler and debugger) to improve the debugging of malware
- Interface DSI to VeriFast (theorem prover based on separation logic) to auto-generate verification conditions for verifying C programs

P. S.: We are looking for a PhD student or PostDoc to join our team in the world-heritage town of Bamberg!

Publications

- **T. Rupprecht, J. H. Boockmann, D. H. White, and G. Lüttgen. DSI: Automated Detection of Dynamic Data Structures in C Programs and Binary Code. In 19th Coll. on Programming Languages and Foundations of Programming (KPS '17), 2017. http://www.kps2017.uni-jena.de/kps2017_proceedings.html**
- T. Rupprecht, X. Chen, D. H. White, J. H. Boockmann, G. Lüttgen, and H. Bos. DSIbin: Identifying dynamic data structures in C/C++ binaries. In 32nd Intl. Conf. on Automated Software Engineering (ASE '17). IEEE/ACM, 2017. Accepted for publication.
- T. Rupprecht, X. Chen, D. H. White, J. T. Mühlberg, H. Bos and G. Lüttgen. POSTER: Identifying dynamic data structures in malware. In 23rd Intl. Conf. on Computer and Communications Security (CCS '16), pp. 1772–1774, ACM, 2016. (DOI)
- D. H. White, T. Rupprecht, and G. Lüttgen. DSI: An evidence-based approach to identify dynamic data structures in C programs. In 25th Intl. Symp. on Software Testing and Analysis (ISSTA '16), pp. 259–269, ACM, 2016. (DOI)
- J. T. Mühlberg, D. H. White, M. Dodds, G. Lüttgen and F. Piessens. Learning assertions to verify linked-list programs. In 13th Intl. Conf. on Software Engineering and Formal Methods (SEFM '15), vol. 9276 of LNCS, pp. 37–52, Springer, 2015. (DOI)
- D. H. White. dsOli: Data structure operation location and identification. In 22nd Intl. Conf. on Program Comprehension (ICPC '14), pp. 48–52, ACM, 2014. (DOI)
- D. H. White and G. Lüttgen. Identifying dynamic data structures by learning evolving patterns in memory. In 19th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13), vol. 7795 of LNCS, pp. 354–369, Springer, 2013. (DOI)

DSI is open source: `https://github.com/uniba-swt`

Backup Slides

Backup Slides

Further DSI Publications

- D. H. White, T. Rupprecht, and G. Lüttgen. dsOli2: Discovery and comprehension of interconnected lists in C programs. In 18th Coll. on Programming Languages and Foundations of Programming (KPS '15), 2015. Proceedings available online at `https://www.complang.tuwien.ac.at/kps2015/proceedings`.

- L. Dietz. Memory snapshot alignment for data structure discovery. Master thesis, U. Bamberg, Germany, January 2017.

- J. Boockmann. Automatic generation of data structure annotations for pointer program verification. Bachelor thesis, U. Bamberg, Germany, October 2016.

- K. Welzel. Heap visualisation using interactive memory graphs. Bachelor thesis, U. Bamberg, Germany, March 2016.

- L. Dietz. Multidimensional repetitive pattern discovery for locating data structure operations. Bachelor thesis, U. Bamberg, Germany, April 2015.

Publications on Related Tools

- Caballero, J., Grieco, G., Marron, M., Lin, Z., and Urbina, D. **ARTISTE**: Automatic generation of hybrid data structure signatures from binary code executions. Tech.Report TR-IMDEA-SW-2012-001, IMDEA Software Institute, Spain, 2012.
- Jung, C. and Clark, N. **DDT**: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In MICRO '09, pp. 56–66. ACM, 2009.
- Haller, I., Slowinska, A., and Bos, H. Scalable data structure detection and classification for C/C++ binaries. Empirical Software Engineering, 21(3):778–810, 2016. (**Mempick**)
- Aftandilian, E. E., Kelley, S., Gramazio, C., Ricci, N., Su, S. L., and Guyer, S. Z. **Heapviz**: Interactive heap visualization for program understanding and debugging. In SOFTVIS '10, pp. 53–62. ACM, 2010.
- Marron, M., Sanchez, C., Su, Z., and Fähndrich, M. Abstracting runtime heaps for program understanding. IEEE Trans. on Software Engineering, 39(6):774–786, 2013. (**Heapdbg**)
- Dudka, K., Peringer, P., and Vojnar, T. Byte-precise verification of low-level list manipulation. In SAS '13, vol. 7935 of LNCS, pp. 215–237. Springer, 2013. (**Predator**)
- Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., and Vojnar, T. Fully automated shape analysis based on forest automata. In CAV '13, vol. 8044 of LNCS, pp. 740–755. Springer, 2013. (**Forester**)

Related Work: Dynamic Shape Analysis I (on x86 binaries)

ARTISTE (Caballero et al., 2012) for signature generation (security applications):

- Examines memory snapshots taken at regular intervals
- Excavates types and constructs points-to graphs
- Merges nodes into region graphs (kind of structural repetition, leading to a folded points-to graph)
- Uses shape predicates to identify data structures

→ Does not employ a fine-granular heap abstraction (unlike DS, see Linux CDLL example)

→ Does not address degenerate shapes (unlike DSI)

→ Does not deal with data structure relationships such as nesting (unlike DSI)

→ Does not consider temporal repetition (unlike DSI)

Related Work: Dynamic Shape Analysis II (on x86 binaries)

MemPick (Haller, Slowinska & Bos, 2015) for reverse engineering (security applications):

- Examines memory snapshots taken during quiescent periods
- Excavates types, constructs points-to graphs, and analyses points-to graphs wrt. overlays and simple, local shape invariants
- Identifies data structures according to a tailored decision tree

→ Can deal with nested and 'mixed' data structures (e.g., linked lists running through trees, as used in topological sorting)

→ Does not employ a fine-granular heap abstraction (unlike DSI, see Linux CDLL example)

→ Is sensitive to inaccuracies in stable shape identification (unlike DSI)

Related Work: Dynamic Shape Analysis III (on x86 binaries)

DDT (Jung & Clark, 2009) for data structure shape and operation identification:

- Examines multiple program executions to heuristically detect interface functions for data structures
- Observes these functions' properties, e.g., node insertions/deletions
- Considers simple, local shape invariants (similar to MemPick)
- Identifies data structure operations based on a tailored decision tree

→ Works well for code that uses standard libraries (e.g., C++ STL)

→ Does not employ a fine-granular heap abstraction (unlike DSI, see Linux CDLL example)

→ Cannot detect relationships such as nesting (unlike DSI)

Related Work: Data Structure Visualization

Heapviz (Aftandilian et al.):

- Dynamic analysis on byte code for visualizing and animating entire runtime heaps
- Folding (or 'merging') of points-to graphs similar to DSI's structural repetition
- Dominator-based layouting scheme that deals with nesting

HeapDbg (Marron et al.):

- Similar to Heapviz, but HeapDbg employs abstract interpretation techniques for merging
- In contrast to Heapviz, HeapDbg also infers shapes with which the visualized graphs are annotated

Related Work: Static Shape Analysis

Predator (Dudka, Peringer & Vojnar) & Forester (Kolik et al.):

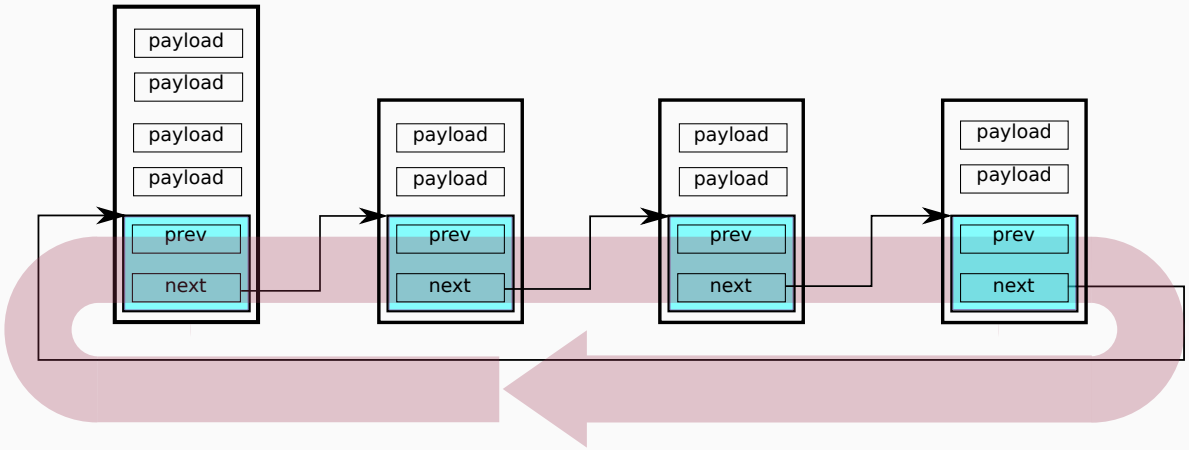
- Focus on program verification (memory safety) rather than program comprehension
- Execute C programs symbolically to infer shape predicates
- Employ symbolic memory graphs and, resp., forest automata

Questions:

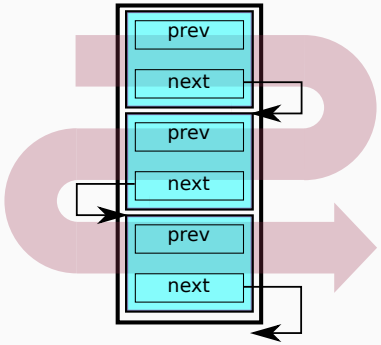
- How similar are Predator/Forester's inferred shape predicates to those that could be generated by DSI?
- How do the assumptions underlying Predator/Forester compare to those of DSI?

DSI Challenges: C Heaps

Nested structs:



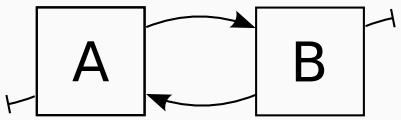
Custom memory allocators:



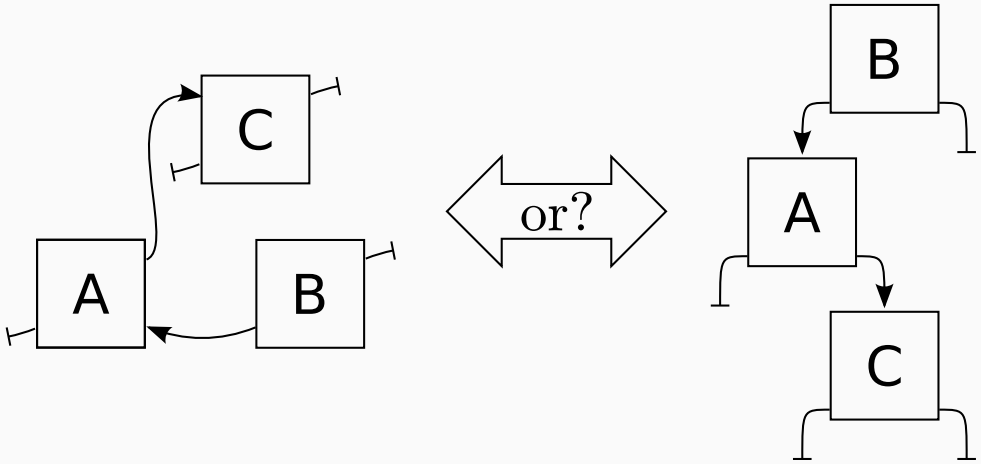
DSI Challenges: Degenerate Shapes

A data structure might temporarily lose its stable shape due to manipulation operations

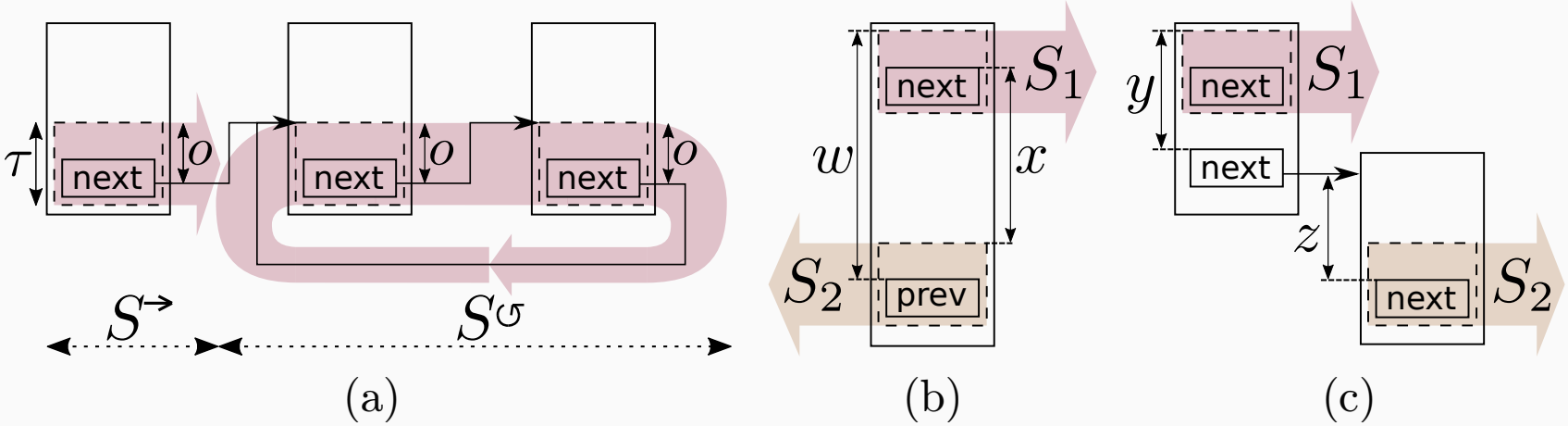
Time t:



Time t+1:

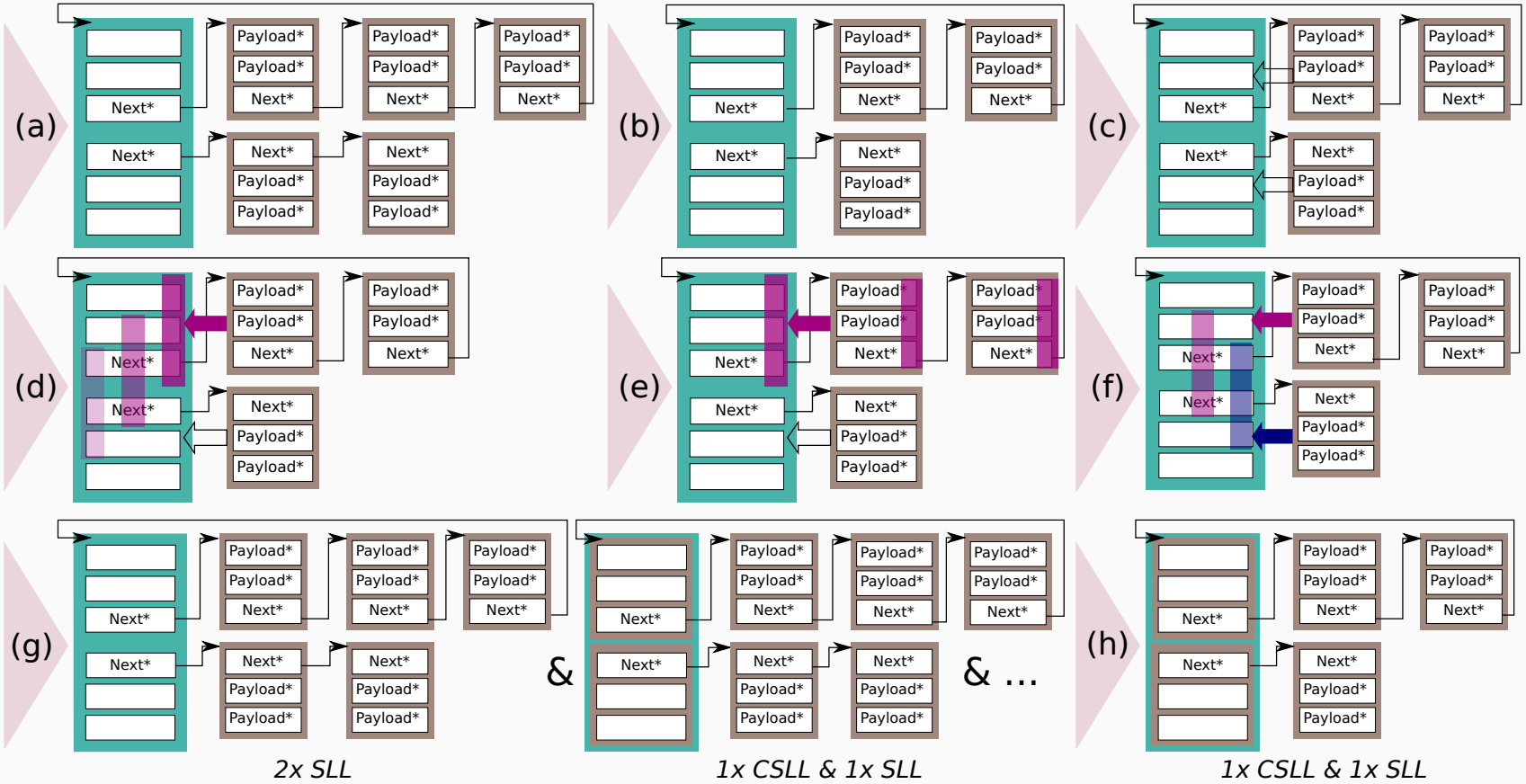


Linkage Conditions

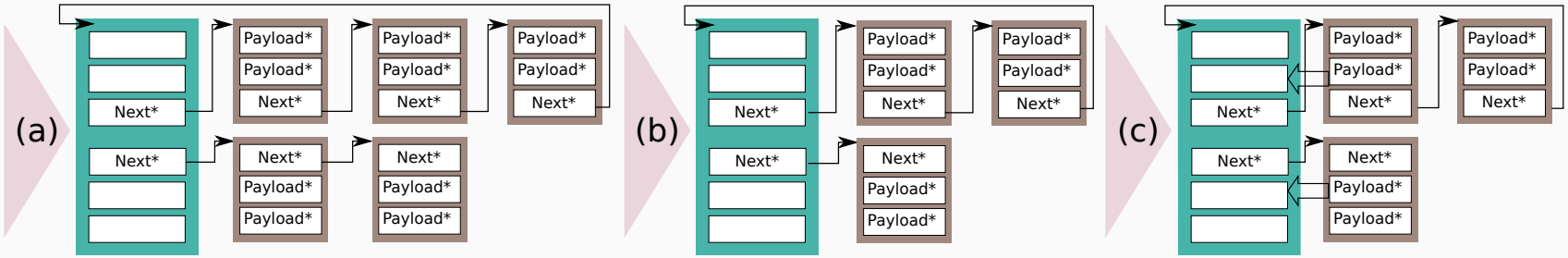


Details of (a) a strand S with linkage condition $S^{LC} = (\tau, o)$ and both linear and cyclic cell sequences, (b) an overlay SC $S_1 \overset{xw}{\longleftrightarrow} S_2$ and (c) an indirect SC $S_1 \overset{yz}{\longrightarrow} S_2$. Memory chunks have black outline, cells are dashed, and strands are indicated with block arrows.

DSIref: Type Refinement (Overview)



DSIref: Type Refinement I

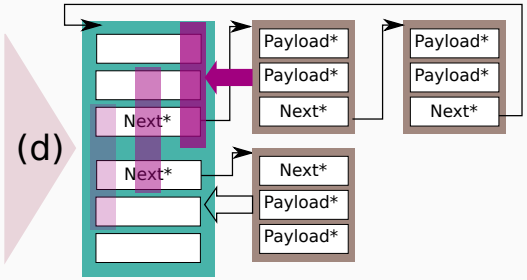


(a) Create sequence of points-to-graphs from program execution (only one shown)

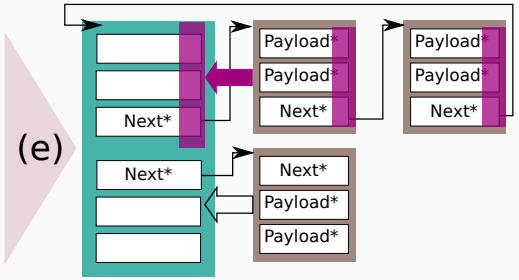
(b) Construct merged type graph capturing pointer connections between types

(c) Exploit pointer connections by mapping type sub-regions (two possibilities shown)

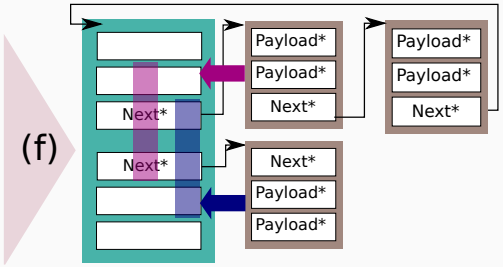
DSIref: Type Refinement II



(d) Observe that multiple interpretations may be possible

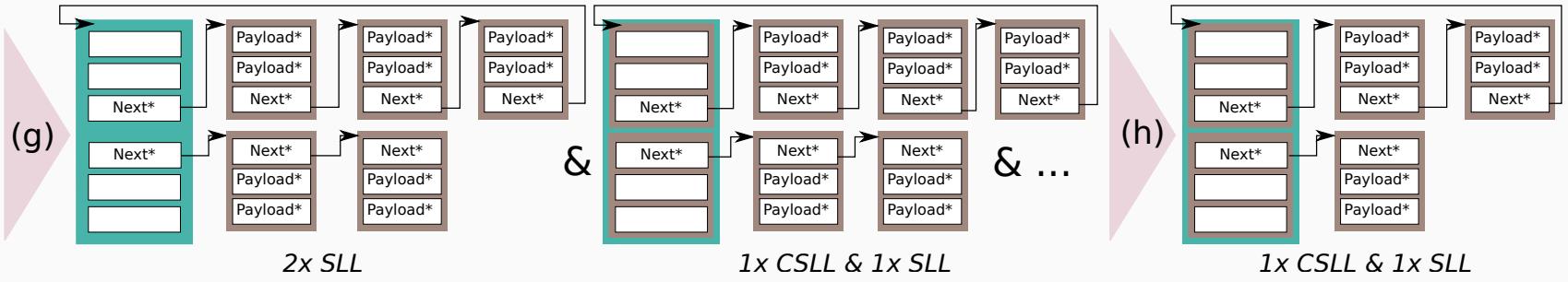


(e) Propagate each interpretation along pointer connections



(f) Rule out inconsistencies

DSIref: Type Refinement III

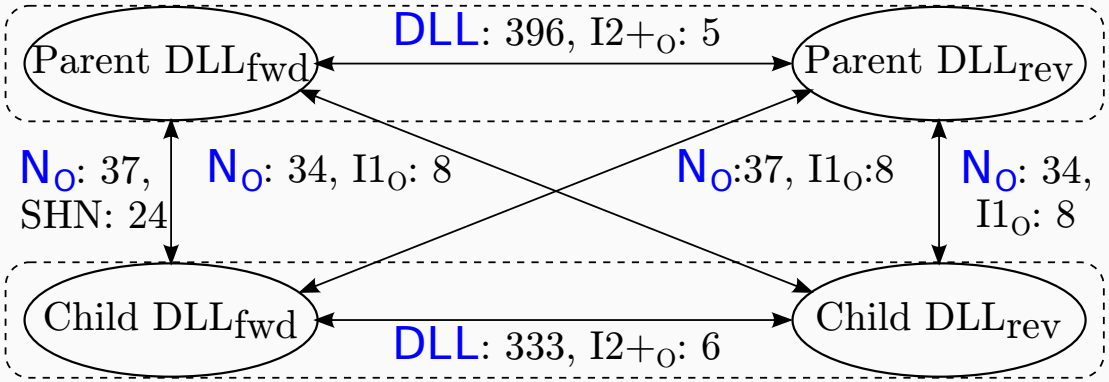


(g) Evaluate remaining interpretations via DSI

(h) Choose the 'best' interpretation in terms of data structure complexity (indicated by merged type graph with resulting label *1xCSLL & 1xSLL*)

DSIsrc Evaluation I

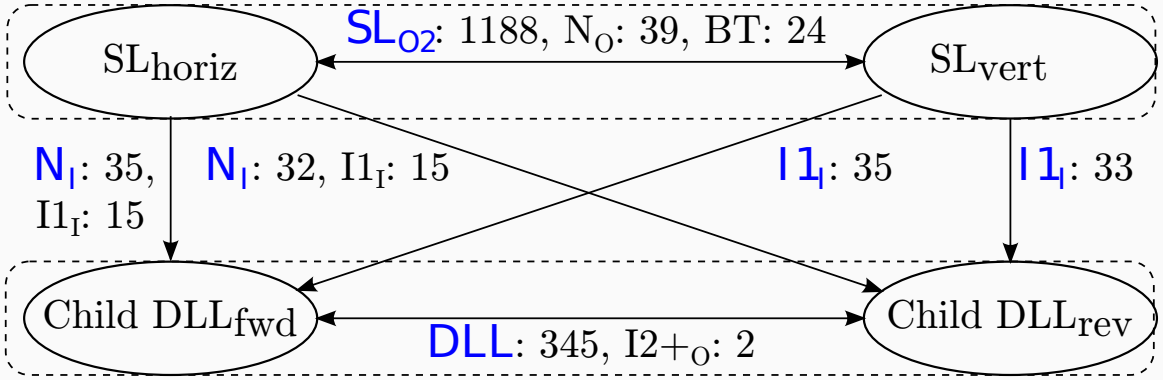
ID	Naming Module Output	Evidence Counts	% Supporting Evidence
tb1	SLL	None since no SCs present	
tb2	DLL	DLL: 1440, $I2+_O$: 220	87%
syn1	CDLL	CDLL: 15, $I2+_O$: 10, DLL: 6	48%
syn2	Binary Tree	BT: 248, N_O: 6, $I1_O$: 3	97%
syn3	SLL + nest. SLL	N_I: 6, $I1_I$: 2	75%
syn4	SLL + nest. SLL	N_O: 10, SHN: 6	63%
syn5	DLL + nest. DLL		Avg. 84%



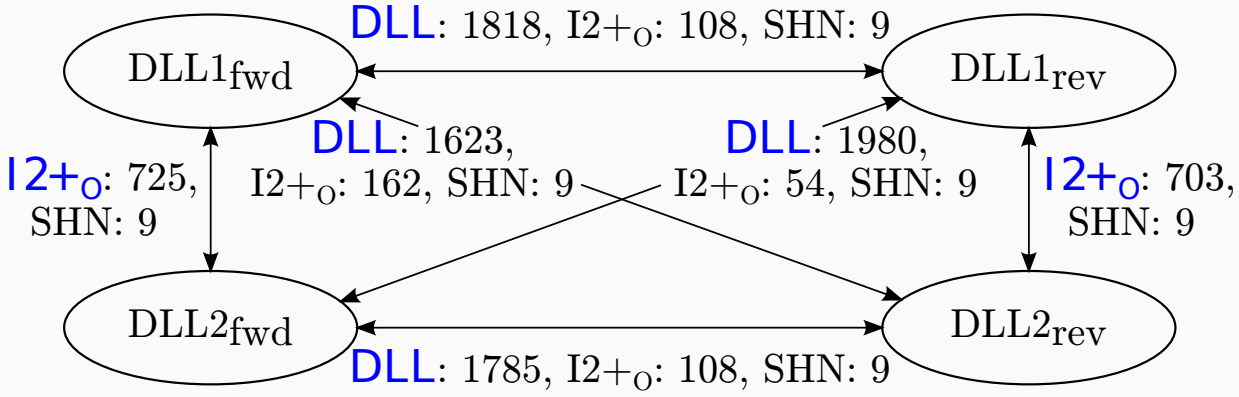
(tb – textbook, syn – synthetic)

DSIsrc Evaluation II

syn6 Skip List + nest. DLL Avg. 89%



lit1 DLL of (DLL, DLL) OR Intersecting(2xDLL) Avg. 96%

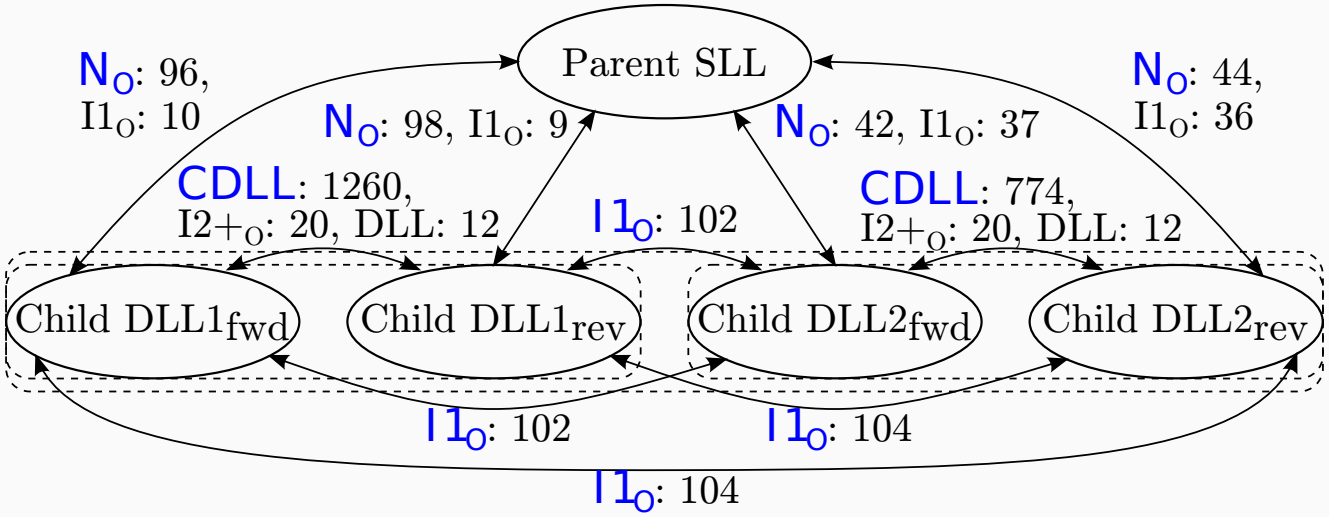


(syn – synthetic, lit – literature)

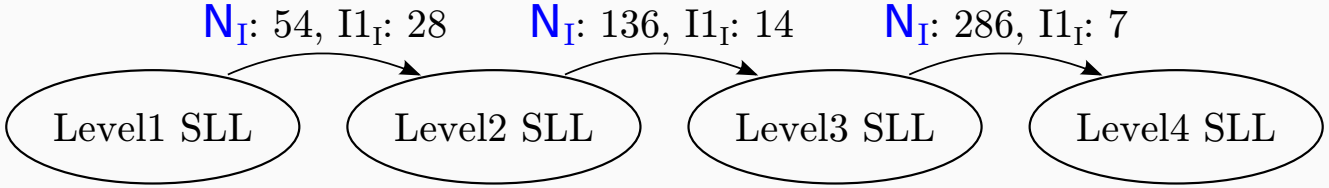
DSIsrc Evaluation III

lit2	Skip List	SL_{02}: 1242, BT: 72, N_O: 58, $I1_O$: 8, SHN: 3	90%

lit3 SLL + nest. Intersecting(2x CDLL) Avg. 88%



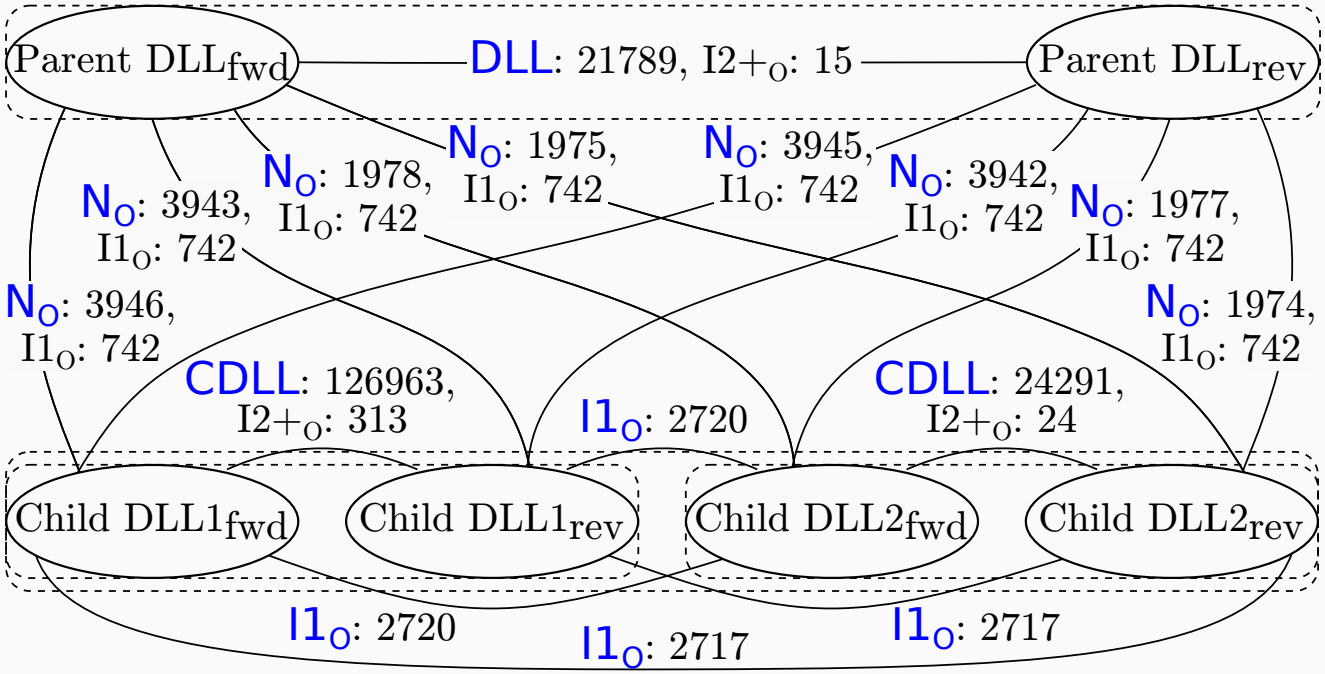
lit4 SLL + nest. SLL + nest. SLL + nest. SLL Avg. 85%



(lit – literature)

DSIsrc Evaluation IV

bash	CDLL	CDLL: 68529, I2+$_O$: 72, DLL: 6	~100%
treeadd	Binary tree	BT: 256	100%
treebnh	Binary tree	BT: 930	100%
libusb	CDLL + nest. Intersecting(2x CDLL)		Avg. 88%



(real-world examples)

Noteworthy Observations wrt. DSIsrc's Evaluation

General observations:

- 13 of the 18 memory structures of our taxonomy are exercised
- Evidence for stable shapes builds up quickly in all examples
- Correct observation is always in the majority

Example-specific observations:

- libusb (7K LOC C) and lit3 are structurally most challenging
- syn1 represents the worst case for DSIcore (short trace, many instances of degenerate shapes)
- lit1 is the only example with an ambiguous reading, revealing a non-obvious memory structure interpretation
  - The forward strand of one DLL can be matched with either DLL's reverse strand
  - This highlights the utility of aggregated strand graphs for analyzing memory structures

Refinement Result for VNC Example

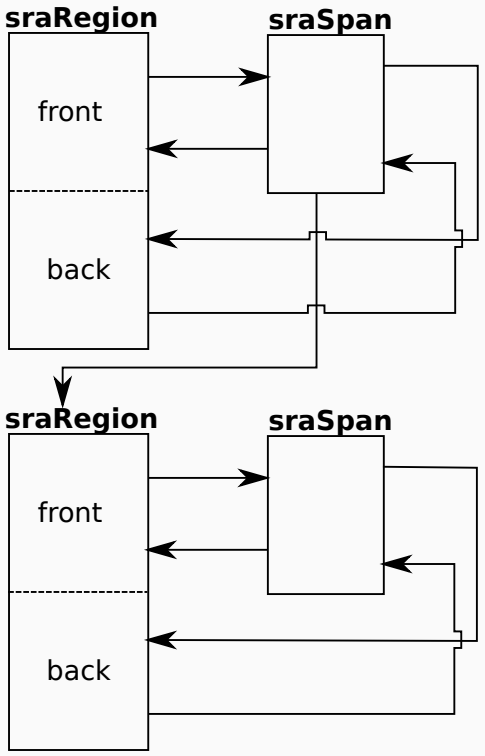
Source	Howard	DSIref

`struct sraSpan {`
`  struct sraSpan *_next;`
`  struct sraSpan *_prev;`
`  int start;`
`  int end;`
`  struct sraRegion *subspan;`
`}`
	`struct {`
	`  0x0: VOID*;`
	`  0x8: VOID*;`
	`  0x10: INT32;`
	`  0x14: INT32;`
	`  0x18: VOID*;`
	`}`
		`struct 1176{`
		`  0x0: VOID*;`
		`  0x8: VOID*;`
		`  0x10: INT32;`
		`  0x14: INT32;`
		`  0x18: VOID*;`
		`}`

`struct sraRegion {`
`  sraSpan front;`
	`struct {`
	`  0x0: VOID*;`
	`  0x8: INT64;`
		`struct{`
		`  0x0: struct 1176{`
		`  0x0: VOID*;`
		`  0x8: VOID*;`
		`  0x10: INT32;`
		`  0x14: INT32;`
		`  0x18: VOID*;`
		`  }`

`  sraSpan back;`	`  0x20: struct{`
	`  0x0: INT64;`
	`  0x8: VOID*;`
	`  };`
		`  0x20: struct 1176{`
		`  0x0: VOID*;`
		`  0x8: VOID*;`
		`  0x10: INT32;`
		`  0x14: INT32;`
		`  0x18: VOID*;`
		`  }`

`}`	`}`	`}`



DSIbin Evaluation

	Code (ground truth)							Naive Howard-DSIcore Pipeline				Sophisticated Howard-DSIref-DSIcore Combination								
expl	**lang**	**DS [LOC / trace length]**	**flat**	**n@h**	**n**	**m**	**h/s**	**DSIbin**	**rec**	**n-d**	**m**	**DSIbin**	**rec**	**n@h-d**	**n-d**	**m**	**nm**	**h/s**	**pr**	**ch hyp**
er-1	C++	**CDLL** [70 / 2009]	n	y	n	y	n	DLL	n	n	y	CDLL	y	y	n	-	y	n	n	DSIref
er-2	C++	**CDLL** [66 / 1474]	n	y	y (p)	y	n	DLL	n	y (p)	y	CDLL	y	y	n (p)	-	y	n	n	DSIref
lit-1	C	**CDLL→2xNo→CDLL,CDLL** [101 / 1326]	n	y	y	y	y	DLL→2xNi→DLL,DLL	n	y	y	CDLL→2xNo→CDLL,CDLL	y	y	y	-	y	y	n	DSIref
lit-2	C	**SLL(→Ni→SLL)x4** [143 / 5667]	n	n	n	n	n	SLL(→Ni→SLL)x4	y	n	n	SLLs with Ni/No	n	n	n	y	-	n	n	(DSIRef)
lit-3	C	**SLo** [179 / 1329]	n	n	n	y	n	SLo	y	n	y	SLo	y	n	n	-	-	n	n	Howard
lit-4	C	**SLL→2xNo→CDLL,CDLL** [90 / 275]	n	y	y	y	n	SLL→2xNi→DLL,DLL	n	y	y	SLL→2xNo→CDLL,CDLL	y	y	y	-	y	n	n	DSIref
lit-5	C	**2 DLLs parallel** [36 / 557]	n	n	n	y	n	4xDLL parallel	y	n	y	2xDLL parallel	n	y	y	-	y	n	n	Howard
r-1	C	**BT** [98 / 5435]	n	n	n	y	n	BT	y	n	y	BT	y	y	n	-	-	n	n	Howard
r-2	C	**BT** [356 / 1635]	n	n	n	y	n	BT	y	n	y	BT	y	n	n	-	-	n	n	Howard
r-3	C	**DLL(5)→Ni→DLL** [712 / 2174]	n	y	y	y	n	DLL (3, no nesting)	n	y	y	DLL (5, with hint on No)	n	y	y	-	y	n	y	(DSIRef)
syn-01	C	**BT** [50 / 1635]	n	n	n	y	n	BT	y	n	y	BT	y	n	n	-	-	n	n	Howard
syn-02	C++	**SLL→No→SLL** [83 / 2298]	n	n	y	y	n	SLL→Ni→SLL	n	y	y	SLL→No→SLL	y	y	y	-	y	n	n	DSIref
syn-03	C++	**SLL (3)** [48 / 161]	n	n	n	y	n	SLL (3)	y	n	y	SLL (3)	y	n	n	-	-	n	n	Howard
syn-04	C	**DLL (10) + I2o+** [111 / 3807]	y	n	y	y	n	DLL (3) + noise	n	n	y	DLL (10) + I2o+	y	n	y	-	y	n	n	DSIref
syn-05	C	**SLL (5)** [64 / 728]	y	n	y	y	n	SLL (2)	n	n	y	SLL (5)	y	n	y	-	-	n	n	DSIref
syn-06	C	**SLL (11)** [72 / 873]	y (pt)	y	y	n	n	nothing	n	y (pt)	n	SLL (11)	y	y	y	-	y	n	n	DSIref
syn-07	C	**CDLL** [214 / 1329]	n	n	y	o	y	DLL	n	y	-	CDLL	y	n	y	-	y	y	n	DSIref
syn-08	C	**DLL→No→DLL** [34 / 204]	y	n	y	y (pt)	n	DLL (only parent)	n	n	y (pt)	DLL→No→DLL	y	y	y	y	y	n	y	DSIref
syn-09	C	**SLo→Ni→DLL** [97 / 761]	n	n	n	y	n	SLo→Ni→DLL	y	n	y	SLo→Ni→DLL	y	n	n	-	-	n	n	Howard
syn-10	C	**SLL→I1o→SLL** [51 / 354]	n	n	n	y (pt)	n	SLL→I1i→SLL	n	n	y (pt)	SLL→No→SLL	n	n	n	y	-	n	n	(DSIRef)
syn-11	C	**SLL (10)** [32 / 585]	n	n	y	n	n	SLL (9)	n	y	n	SLL (10)	y	n	y	-	y	n	n	DSIref
syn-12	C	**SLL (12)** [49 / 191]	n	n	n	y (pt)	n	SLL (11)	n	n	y (pt)	SLL (12)	y	n	n	y	-	n	n	DSIref
syn-13	C	**SLL (6)** [53 / 72]	n	n	n	n	n	nothing	n	n	n	SLL (6)	y	n	n	y	-	n	n	DSIref
syn-14	C	**SLL→No→SLL** [47 / 352]	y	n	y	o	n	SLL→Ni→SLL	n	n	-	SLL→No→SLL	y	n	y	-	y	n	n	DSIref
syn-15	C	**SLL→No→SLL** [45 / 796]	n	n	n	y	n	SLL→No→SLL	y	n	y	SLL→No→SLL	y	n	n	-	-	n	n	Howard
syn-16	C	**SLL (5)** [29 / 329]	n	n	n	o	y	SLL (4)	n	n	-	SLL (5)	y	n	n	-	-	y	n	DSIref
tb-1	C	**SLL (21)** [130 / 925]	n	n	n	n	n	SLL (20)	n	n	n	SLL (21)	y	n	n	y	-	n	y	DSIref
tb-2	C	**SLL (11)** [132 / 367]	n	n	n	n	n	SLL (10)	n	n	n	SLL (11)	y	n	n	y	-	n	y	DSIref
tb-3	C	**DLL** [217 / 1174]	n	n	y (p)	y	n	DLL	y	n (p)	y	DLL	y	n	n (p)	-	-	n	n	Howard
tb-4	C	**SLL (11)** [105 / 869]	n	n	n	y	n	SLL (10)	n	n	n	SLL (10)	n	n	n	n	-	n	n	(Howard)

Symbol explanation: `flat` flattened member access, `n@h` nesting at head, `n` nesting, `m` type merge, `h/s` DS distributed between heap and stack, `rec` DS recognized, `n-d` nesting detected, `n-m` nested types merged, `pr` primitive types refined, `ch hyp` chosen hypothesis

Noteworthy Observations wrt. DSIbin's Evaluation

General observations:

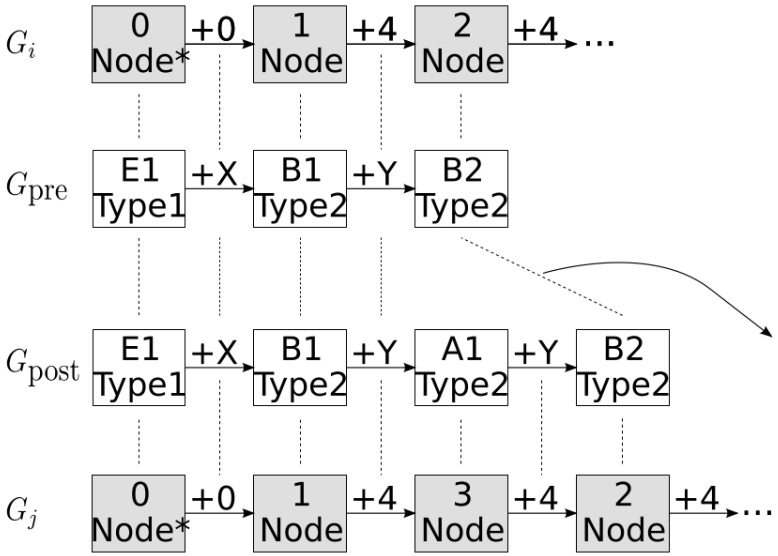
- 30 examples (up to 5.5K events, 156 generated type hypotheses)
  - Execution time in the order of (tens of) minutes due to DSIcore
- DSIref is key for analyzing binaries
  - Naive Howard-DSIcore pipeline correctly identifies 10 of 30 examples
  - Howard-DSIref-DSIcore correctly identifies 26 of 30 examples

Example-specific observations:

- syn-08 shows flattened access of nested head node of child DLL inside parent DLL, so that Howard misses the child list's head
  - DSIcore then recognizes indirect nesting rather than overlay nesting
- lit-2 consists of multi-level SLLs where all types are different but binary compatible, so that Howard wrongly merges types
  - DSIcore then detects some overlay nesting as indirect nesting
- r-3 leads DSIcore to misinterpret indirection as strands of length 2
  - DSIcore should wait until strands have further evolved, before folding

Verification Back End (VeriFast Annotation Generation)

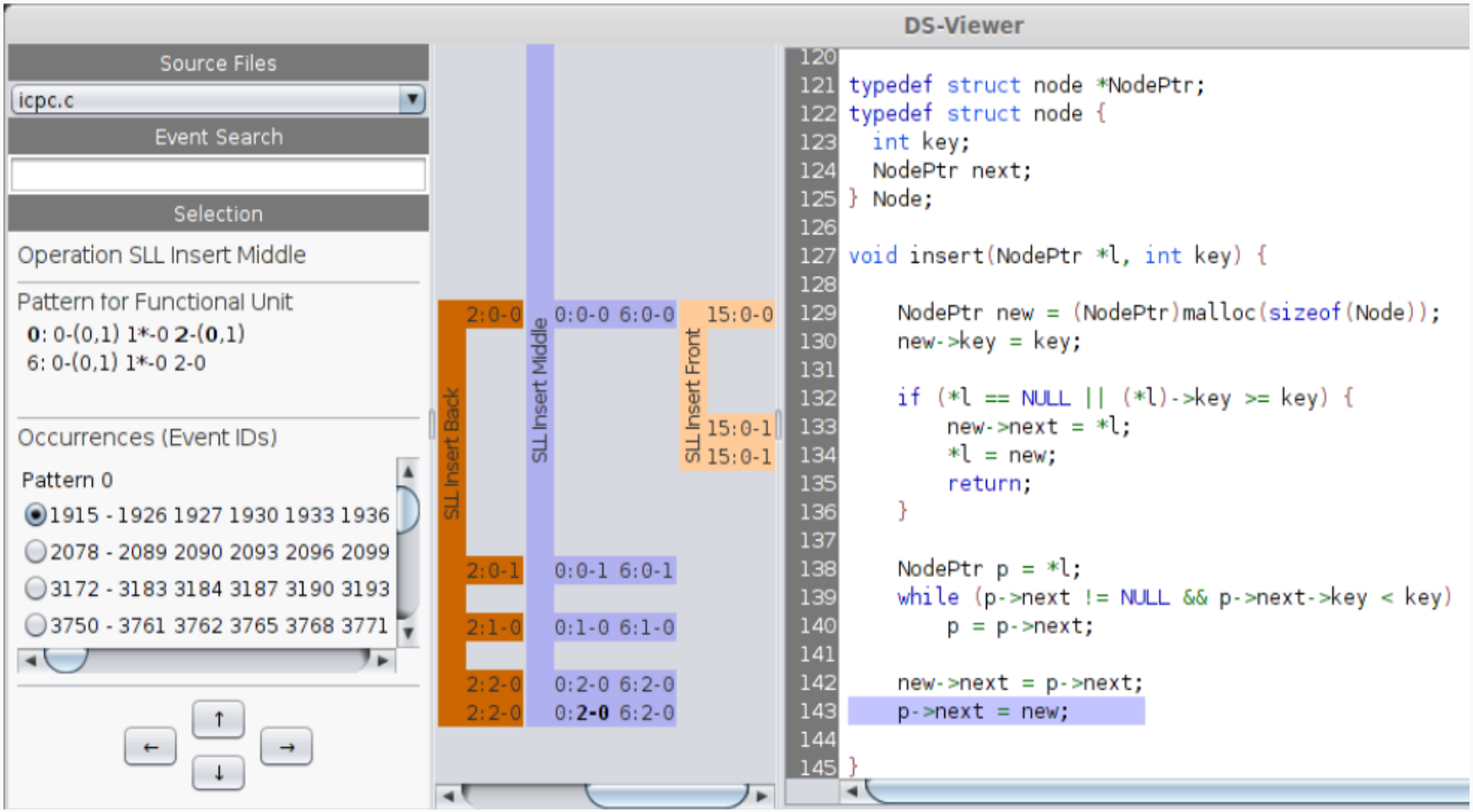
```
1  typedef struct Node *Stack;
2
3  struct Node {
4    ElementType  Element;
5    struct Node  *Next;
6  };
7
8  /*@ predicate SLLNodes_Node(struct↩
       Node *node, int count) =
9    node == 0 ? count==0 : 0<count/
10   &*& node->Element |-> _/
11   &*& node->Next |-> ?next/
12   &*& malloc_block_Node(node)/
13   &*& Nodes_0(next, count-1); @*/
```




```
14 /*@ predicate SLL_Node(struct Node↩
       *list, int count) =
15   0 <= count
16   &*& list->Element |-> _
17   &*& list->Next |-> ?head
18   &*& malloc_block_Node(list)
19   &*& Nodes_0(head, count);  @*/
20
21 void push(ElementType X, Stack S)
22 //@ requires SLL_Node(S, ?n_0);
23 //@ ensures SLL_Node(S, n_0+1);
24 {
25   Stack TmpCell;
26   TmpCell = malloc(
27     sizeof(struct Node));
28   if( TmpCell == NULL ) {
29     printf( "Out of space!" );
30     exit(EXIT_FAILURE);
31   }
32   else {
33 //@ open SLL_Node(S, n_0);
34     TmpCell->Element = X;
35     TmpCell->Next = S->Next;
36     S->Next = TmpCell;
37 //@ close SLLNodes_Node(S->Next, ↩
       n_0+1);
38 //@ close SLL_Node(S, n_0+1);
39   }
40 }
```

tid ttype	Template graph vertex $v \in \mathcal{V}^T$ with attributes v.tid & v.ttype	cid ctype	Points-to graph vertex $v \in \mathcal{V}^P$ with attributes v.cid & v.ctype

Operation Detection Back End



(Figure shows prototype, taken from DSI's predecessor tool dsOli)