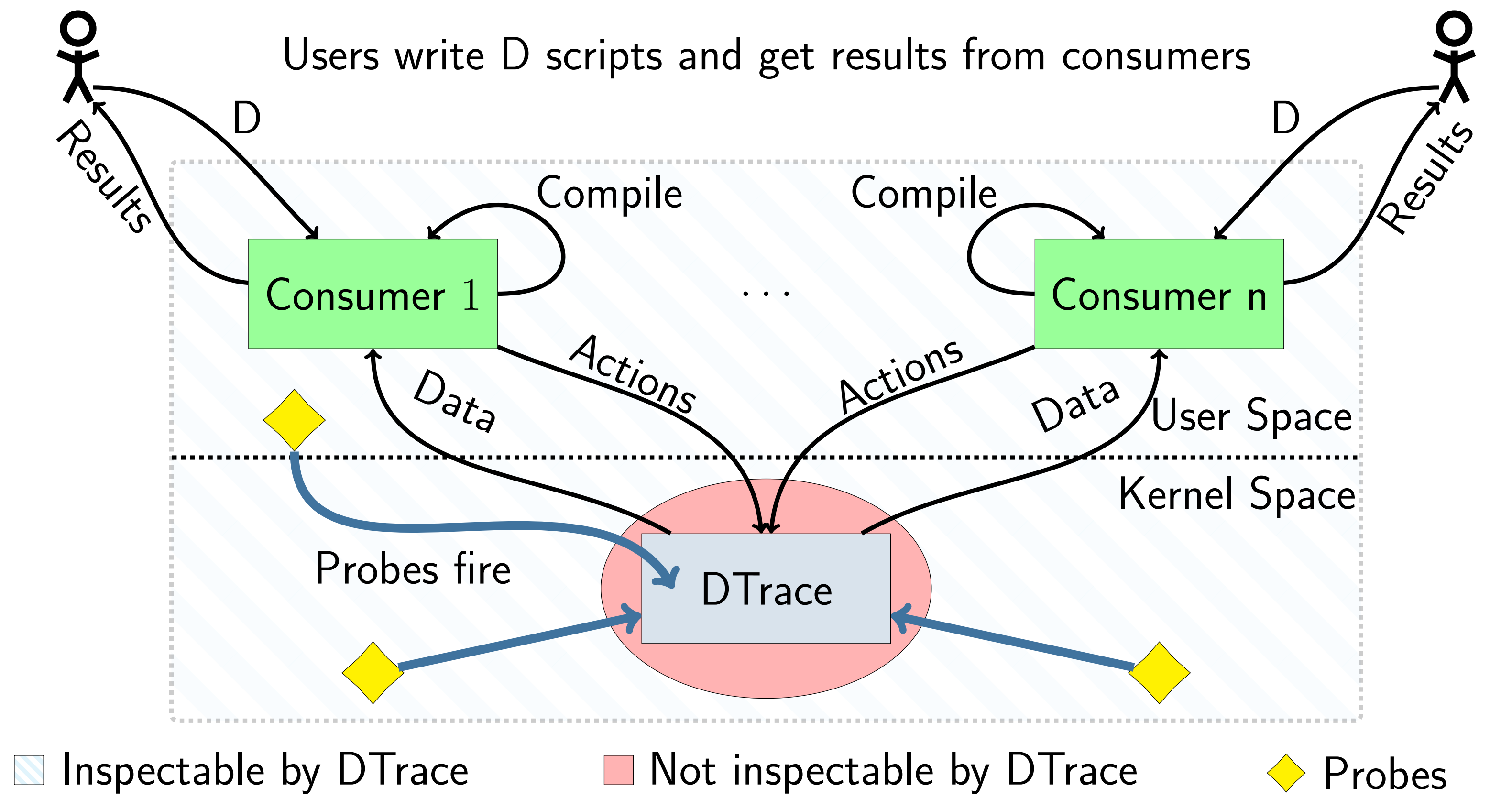
Formal Semantics for the DTrace Tracing System

Student: Domagoj Stolfa
Advisors: Robert N. M. Watson, Alan Mycroft

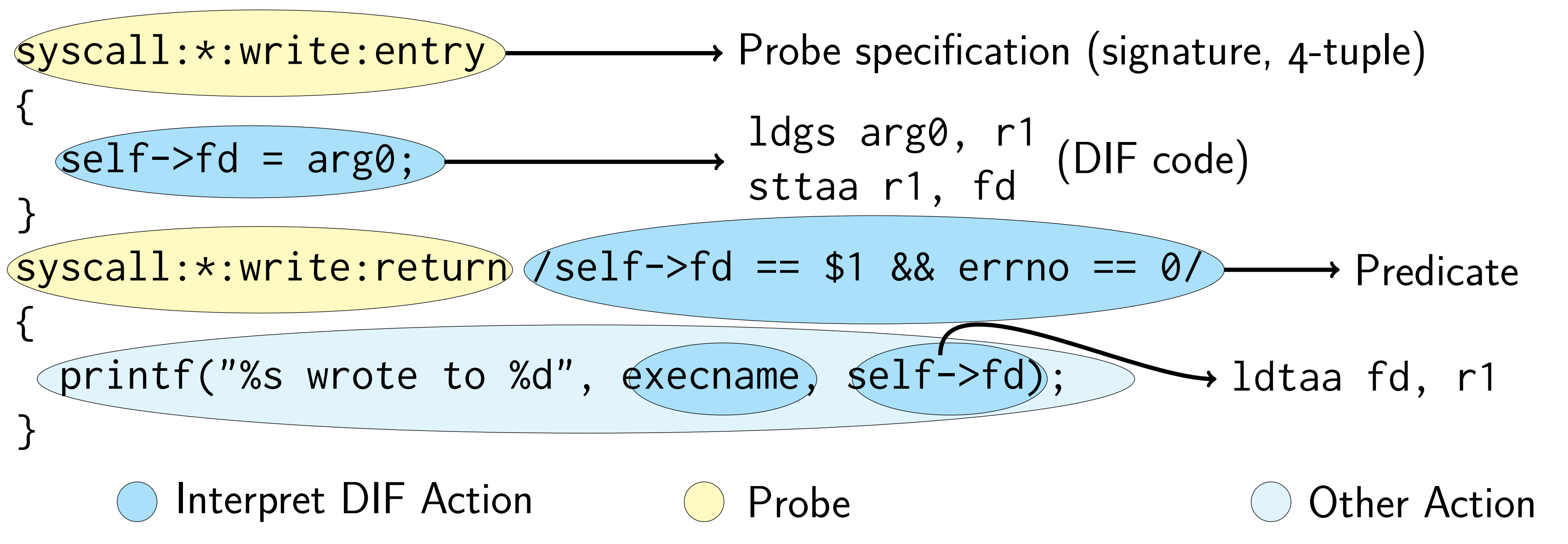
University of Cambridge

Background and Motivation

Tracing systems, tools originally designed to debug and profile the performance of large systems with minimal disruption, are now used increasingly for behaviour monitoring [1, 5] and security auditing [2, 6]. Because of security-critical use and complex semantics, a formal description of such a system is required. In this work we target the most widely deployed tracing system, DTrace, and formalise its semantics in HOL4 [3].



DTrace accepts *scripts* written in D (below); these express *probes* and are compiled to bytecode (*actions* and *DTrace Intermediate Format* (DIF)) in *consumers*. The following script tracks write operations to a Unix file descriptor (files, standard output, socket, etc.):



```
syscall:*:write:entry
{
  self->fd = arg0;
}
syscall:*:write:return /self->fd == $1 && errno == 0/
{
  printf("%s wrote to %d", execname, self->fd);
}
```

```
ldgs arg0, r1
sttaa r1, fd
```

```
ldtaa fd, r1
```

The Formal Model

$P, Q, R ::=$	Semantic Probes (S-Probes)	$\alpha ::=$	Transition labels
$\mid \mathbf{0}$	null probe	$\mid d, t : \mathbf{W}\ \mathsf{x}[\Theta] = V$	write TLS
$\mid A$	sequence of DTrace actions	$\mid d, t : \mathbf{R}\ \mathsf{x}[\Theta] = V$	read TLS
$\mid P \mid Q$	parallel composition	$\mid d : \mathbf{W}\ \mathsf{x}[\Theta] = V$	write global
$\mid P.Q$	sequential composition	$\mid d : \mathbf{R}\ \mathsf{x}[\Theta] = V$	read global
$\mid !P$	replication	$\mid d : \tau$	internal

DTrace probes fire concurrently (even self-concurrently) and can non-deterministically discard data and stop executing any further code as a result of a failure (we denote this as FAIL in our presentation). We model them as S-Probes (concurrent processes) and present their run-time semantics as a labelled transition system of form $\vdash P \xrightarrow{\alpha} P'$.

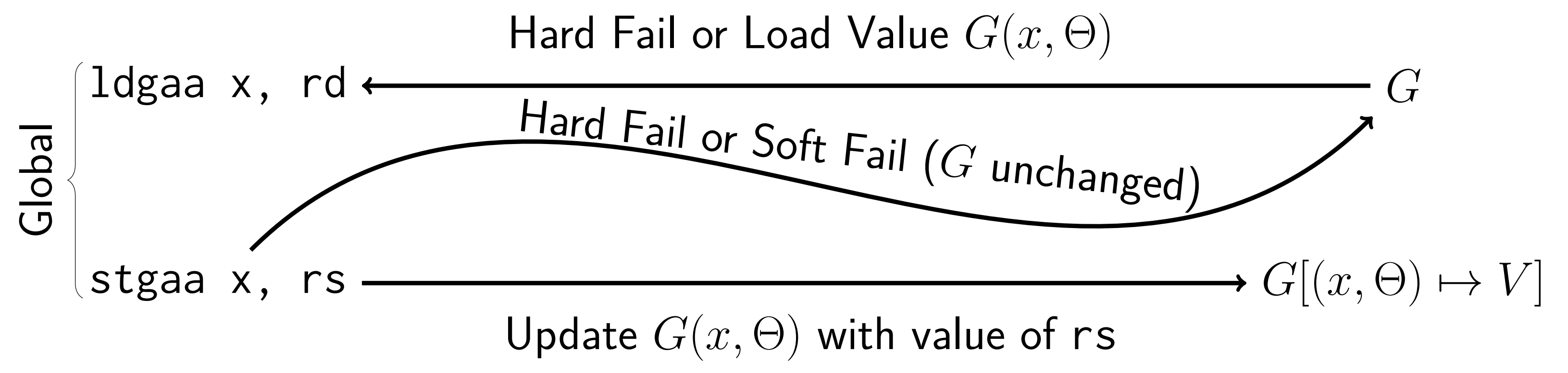
$$\frac{\vdash P_1 \xrightarrow{\alpha} P_1'}{\vdash P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2} \qquad \frac{\vdash P_1 \xrightarrow{\alpha} P_1'}{\vdash P_1.P_2 \xrightarrow{\alpha} P_1'.P_2}$$

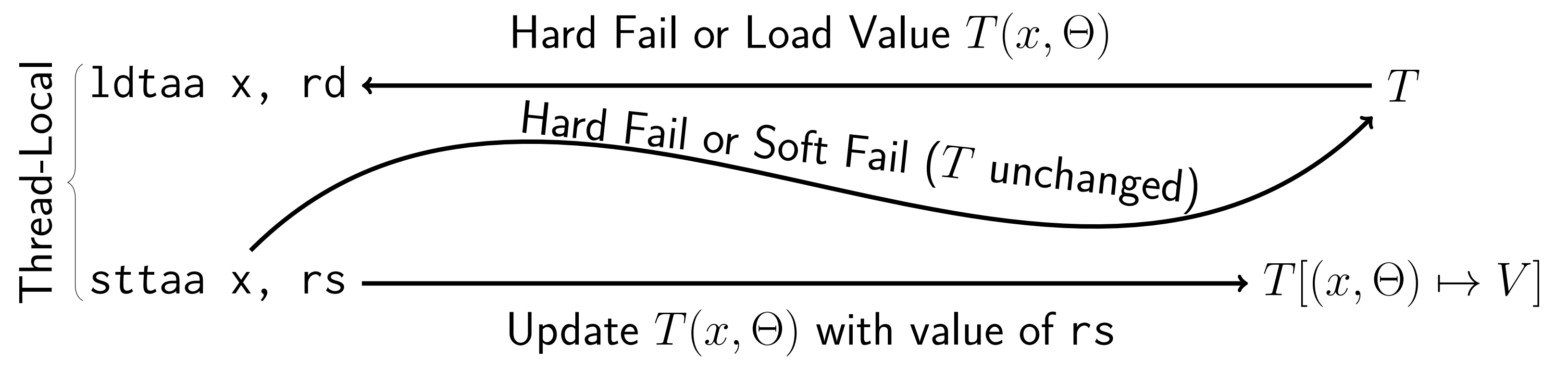
$$\frac{\vdash A \xrightarrow{\alpha} A'}{\vdash A \xrightarrow{\alpha} A'} \qquad \frac{\vdash A \xrightarrow{\alpha} \mathsf{FAIL}}{\vdash A \xrightarrow{\alpha} \mathbf{0}}$$

$$\frac{\vdash P \equiv P' \quad \vdash P' \xrightarrow{\alpha} Q' \quad \vdash Q \equiv Q'}{\vdash P \xrightarrow{\alpha} Q}$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$
$$P \mid Q \equiv Q \mid P \qquad !P \equiv P \mid !P$$
$$P.(Q.R) \equiv (P.Q).R$$
$$P \mid \mathbf{0} \equiv P \qquad \mathbf{0}.P \equiv P$$

Dynamic D variables (compiled to DIF variables in probes) have complex semantics (see forthcoming paper for full definition) and can be either global or thread-local. Reading an unmapped variable returns the value 0, while allocation is performed upon writing a value to a variable the first time. Allocation may fail if name resolution (e.g. an associative array) causes a page fault (hard failure, stops executing any further code in the probe) or if there is no memory to allocate a new variable (soft failure).





Insights — DTrace as a Protocol

$P, Q, R ::=$	S-Probes
$\mid \dots$	
$\mid (\nu c)P$	new channel c, $c \in fv(P)$
$\mid c?.P$	receive on c
$\mid c!.P$	send on c
$\mid c = \bar{v}$	static channel value
$\mid (c, D)?.P$	receive on (c, D)
$\mid (c, D)!.P$	send on (c, D)
$\mid (c, \Theta) = \bar{v}$	dynamic channel value

$$\frac{\vdash P \xrightarrow{d:\mathbf{W}\ c=v_2} P'}{\vdash c!.P \mid c = v_1 \rightarrow P' \mid c = v_2}$$

$$\frac{\vdash P \xrightarrow{d:\mathbf{R}\ c=v_1} P'}{\vdash c?.P \mid c = v_1 \rightarrow P' \mid c = v_1}$$

$$\frac{\vdash P \xrightarrow{d:\mathbf{W}\ (c,\Theta)=v_2} P' \quad \vdash D \downarrow \Theta}{\vdash (c, D)!.P \mid (c, \Theta) = v_1 \rightarrow P' \mid (c, \Theta) = v_2}$$

$$\frac{\vdash P \xrightarrow{d:\mathbf{R}\ (c,\Theta)=v_1} P' \quad \vdash D \downarrow \Theta}{\vdash (c, D)?.P \mid (c, \Theta) = v_1 \rightarrow P' \mid (c, \Theta) = v_1}$$

We observe that loads and stores to DIF variables can be modelled as communication. Our grammar and inference rules are inspired by Honda, et. al [4]. We use $\downarrow$ to denote evaluation of a sequence of DIF actions (D) to a list of values (Θ). Each of the variables in a D script is represented as a named channel. We split channels into two distinct categories:

1. **Static channels** – used for variables that can be named at compile-time. They are specified syntactically and are not subject to run-time name resolution failures.
2. **Dynamic channels** – used for variables with least one run-time dependency on their name (e.g. associative arrays, thread-local variables).

Let P be an arbitrary S-Probe. We define a happens-before ($\sqsubseteq_s^P$) relation dependent on a static channel s and P to be:

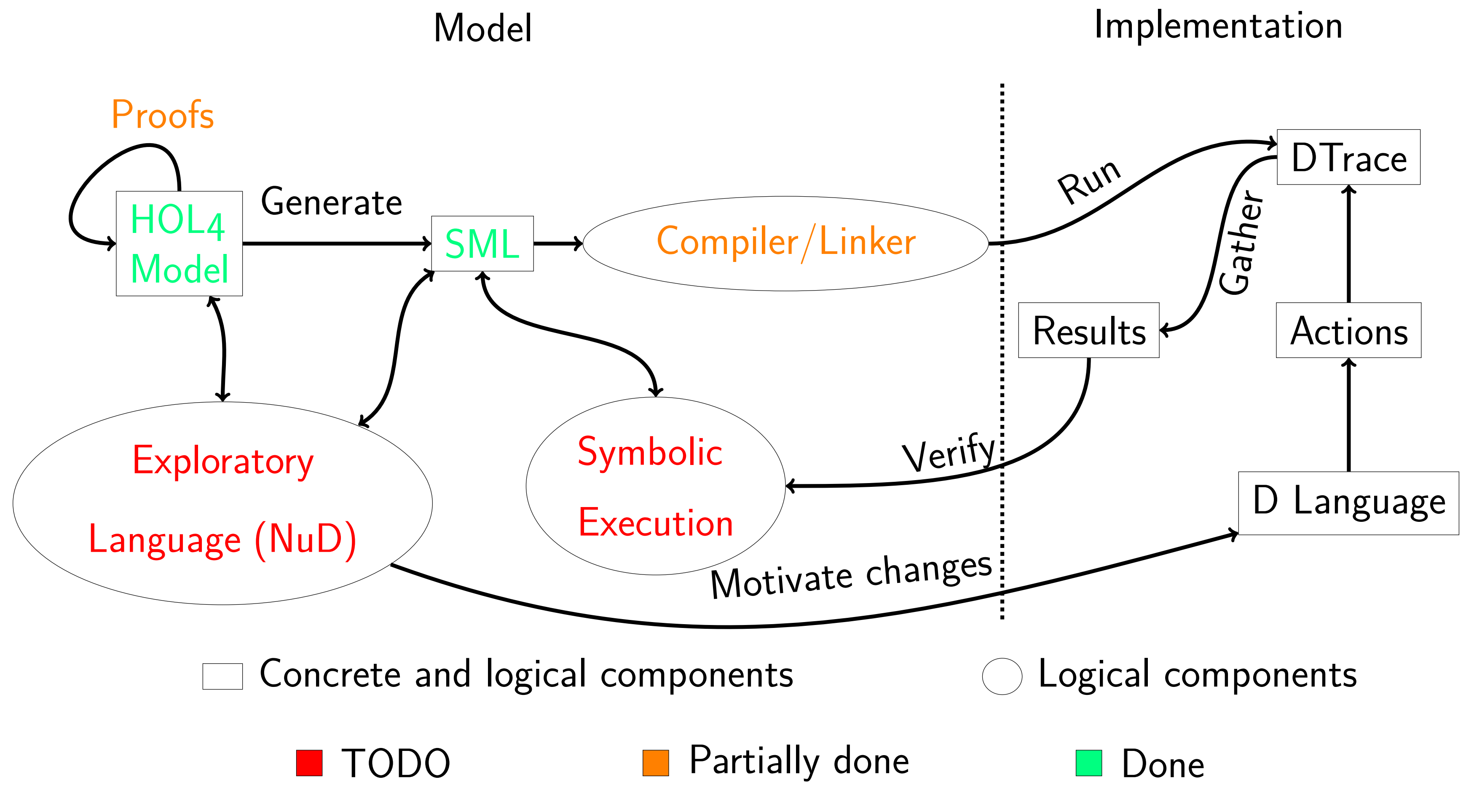
$$p_1 \sqsubseteq_s^P p_2 \iff s!.p_1 \mid s?.p_2 \vee (\exists\, p_3 \in P \,.\, s!.p_1.p_3.s?.p_2)$$

We use that to define data races with respect to s and P as $p_1 \sqsubseteq_s^P p_2 \wedge p_2 \sqsubseteq_s^P p_1$ which we can statically detect. However, with dynamic channels we face the following challenges:

1. **Name resolution failure**: Non-deterministic selection of publishing a value or ending the protocol for a given probe.
2. **Out-of-memory condition**: Non-deterministic addition of a branch.
3. **Programmer assumptions** about the run-time portions of channel names, making it difficult to statically detect data races.

Using the Formal Model as an Implementation

While the formal model itself gives a better understanding of DTrace, we have started leveraging the existing HOL4 implementation of the formal model in order to generate executable Standard ML. We plan to use that in order to implement symbolic execution and use that as a test oracle for DTrace implementations. Moreover, we hope to implement NuD, a safer exploratory language as an alternative to D which is more amenable to automated reasoning. We have started formulating a type system inspired by multi-party session types for which NuD will serve as an implementation.



Acknowledgements and References

This work is part of the CADETS Project sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-15-C-7558. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

[1] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, pages 85–90, 2003.

[2] T. Beauchamp and D. Weston. Dtrace: The reverse engineer's unexpected swiss army knife. *Blackhat Europe*, 2008.

[3] M. J. Gordon and T. F. Melham. Introduction to hol a theorem proving environment for higher order logic. 1993.

[4] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008.

[5] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, 2010.

[6] M. Spainhower. Feasibility analysis of dtrace for rootkit detection. 2008.