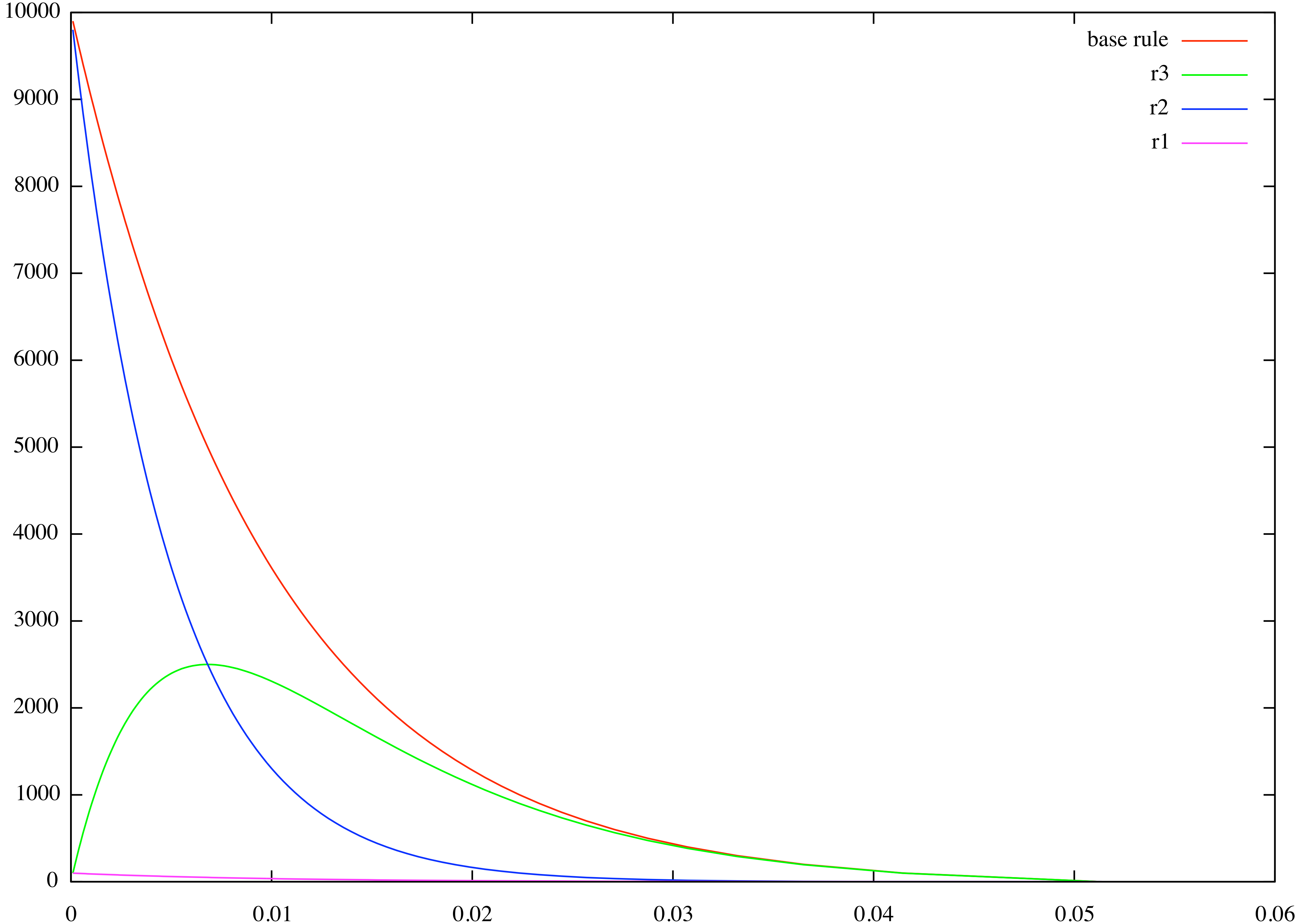
Refinement of rules

- *Don't care, don't write* means that a rule only describes *part* of its actual context
  - many actual contexts are possible (these are all the possible *instances* of the rule)
- We refine a rule by making explicit some (or all) of that missing context
  - allows us to examine the relative importance of different instances
  - allows us to fill in "missing context"

Relative importance



Relative importance (2)

- For this to make sense, we need to ensure that the set of refined rules has the same behaviour as the original rule...
- A refinement is *neutral* if this is the case
  - the appropriate rate constants can be calculated statically
  - the neutral refinement gives a “base line” from which kinetic pertubations can be made...

“Missing context”

- Suppose we have the rule (from before)

`A(b), B(a) -> A(b!1), B(a!1) @` k

- Then somebody tells us that this binding is far more likely if `B` is already bound to C...

`A(b), B(a,c) -> A(b!1), B(a!1,c) @` k_1

`A(b), B(a,c!_) -> A(b!1), B(a!1,c!_) @` k_2

- Neutral refinement: $k = k_1 = k_2$
- Our desired *non*-neutral refinement: $k_1 << k_2$

Formally...

- Describe a “soup” as a *graph-with-sites (gws)*
- An instance of a rule is a gws-homomorphism from the rule’s LHS to the current soup
- For any choice of “additional context”, we can find a set (actually multiset) of refinements
  - divides the rule into *disjoint* subcases (but cases may be repeated...)
  - *n* repeated cases scales the rate constant by *n*

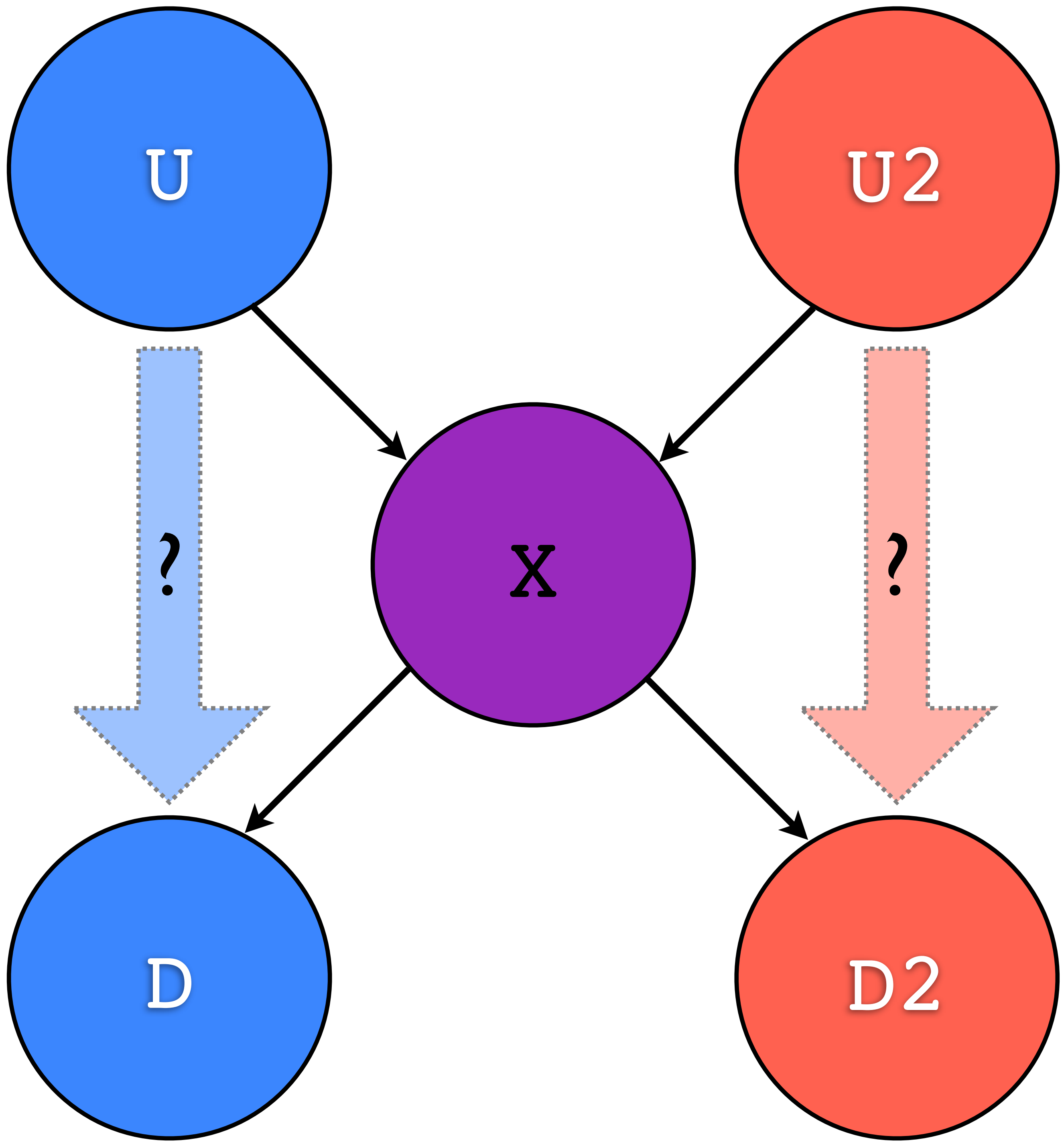
Refinement summary

- Completely formal notion of rule refinement
  - can be used to "split" a rule into several pieces and evaluate their relative contributions
- Neutral refinements assign rates to the rule pieces that leave behaviour unchanged
- Kinetic refinement then modifies this, e.g.
  - an enzyme may have lower affinity for its product than its substrate
  - GPCRs (bond strength between α and $\beta\gamma$)

Refinement (2)

- Another view... refinements reflect small "pertubations" of a rule:
  - expose some previously hidden bit of context
  - use non-neutral kinetics to modify behaviour
- A plausible mechanism by which a signalling network could be subjected to selection
  - e.g. different affinities of splice variants

A specificity puzzle



A simple cascade:
`U -> X -> D`

- `U(s), X(s~u) -> U(s!1), X(s~u!1)`

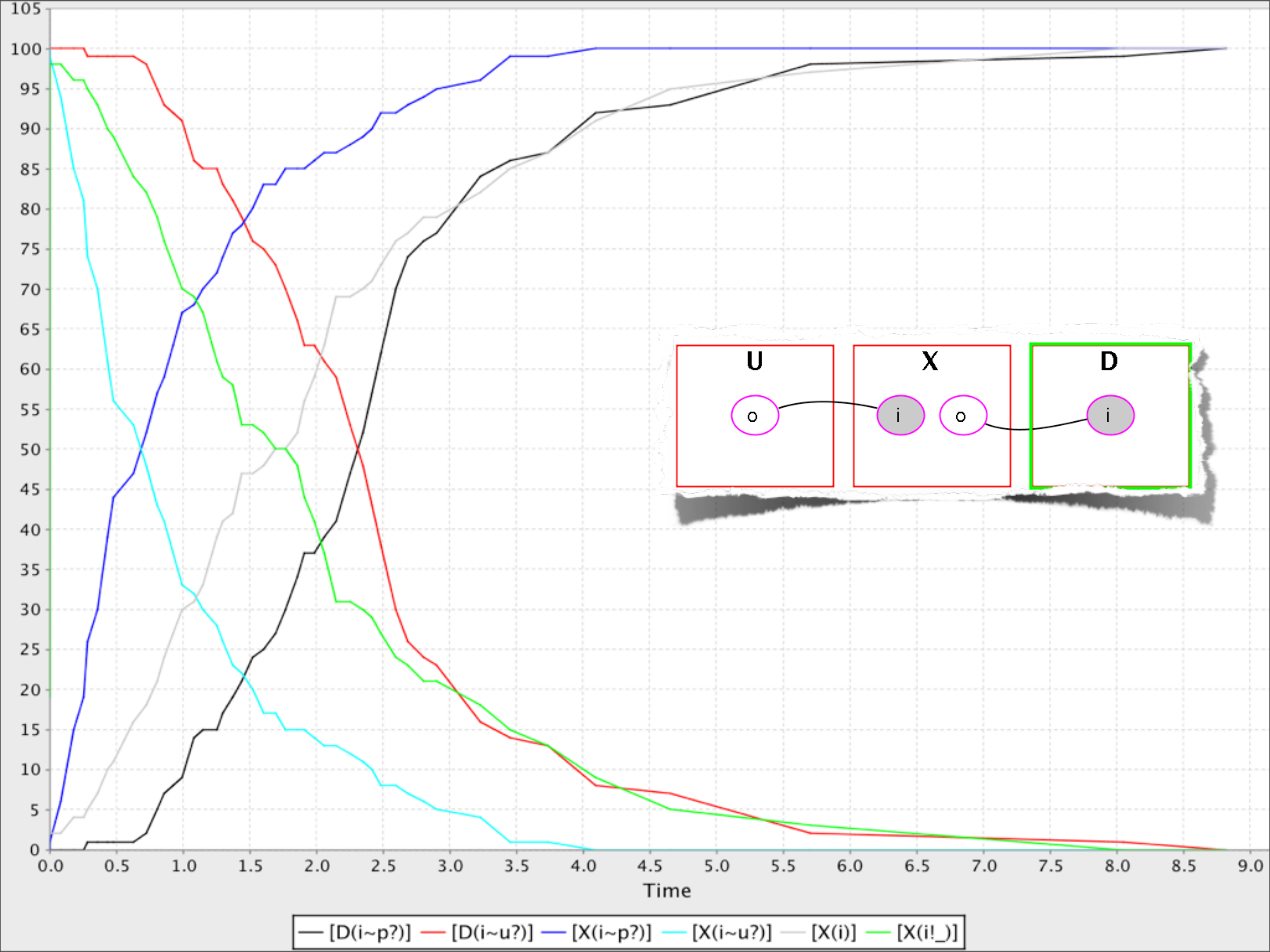
  `U(s!1), X(s!1) -> U(s), X(s)`

  `U(s!1), X(s~u!1) -> U(s!1), X(s~p!1)`

- `X(s~p?,d), D(s~u) -> X(s~p?,d!1), D(s~u!1)`

  `X(d!1), D(s!1) -> X(d), D(s)`

  `X(d!1), D(s~u!1) -> X(d!1), D(s~p!1)`



105
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5
9.0
Time
U
X
D
o
i
o
i
[D(i~p?)]
[D(i~u?)]
[X(i~p?)]
[X(i~u?)]
[X(i)]
[X(i!_)]

Refining U:
`U/U2 -> X -> D`

- refine the `U/I` *off* rule to:

```
U(s!1,u~1), X(s!1) -> U(s,u~1), X(s) @ lo

U(s!1,u~2), X(s!1) -> U(s,u~2), X(s) @ hi
```

- So U now exists in two distinct forms with different affinities for the common substrate `X`

Refining D:

`U/U2 -> X -> D/D2`

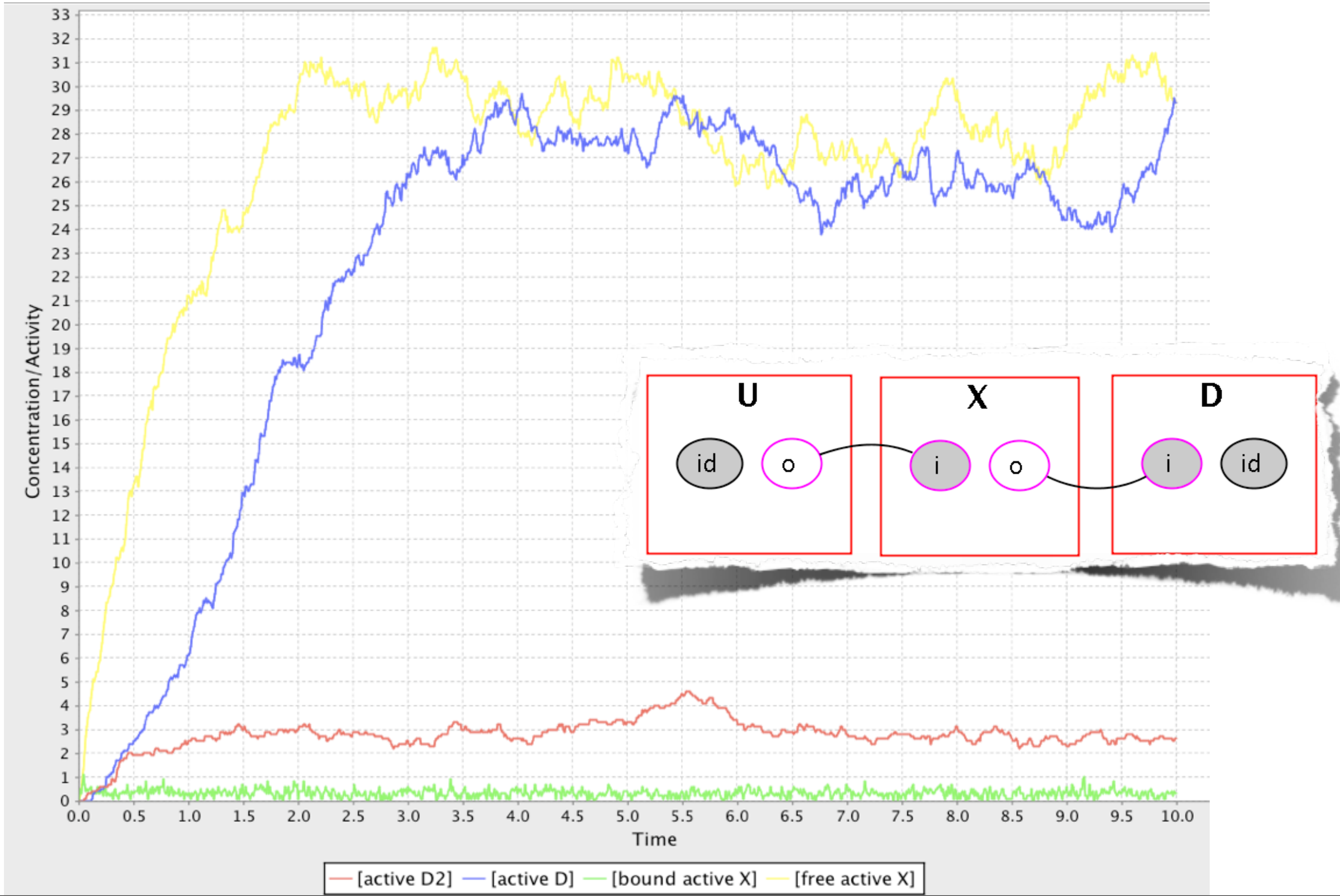
- refine the `I/D` *on* rule to:

```
X(s~p,d), D(s~u,d~1) -> X(s~p,d!1), D(s~u!1,d~1)

X(s~p!_,d), D(s~u,d~2) -> X(s~p,d!1), D(s~u!1,d~2)
```

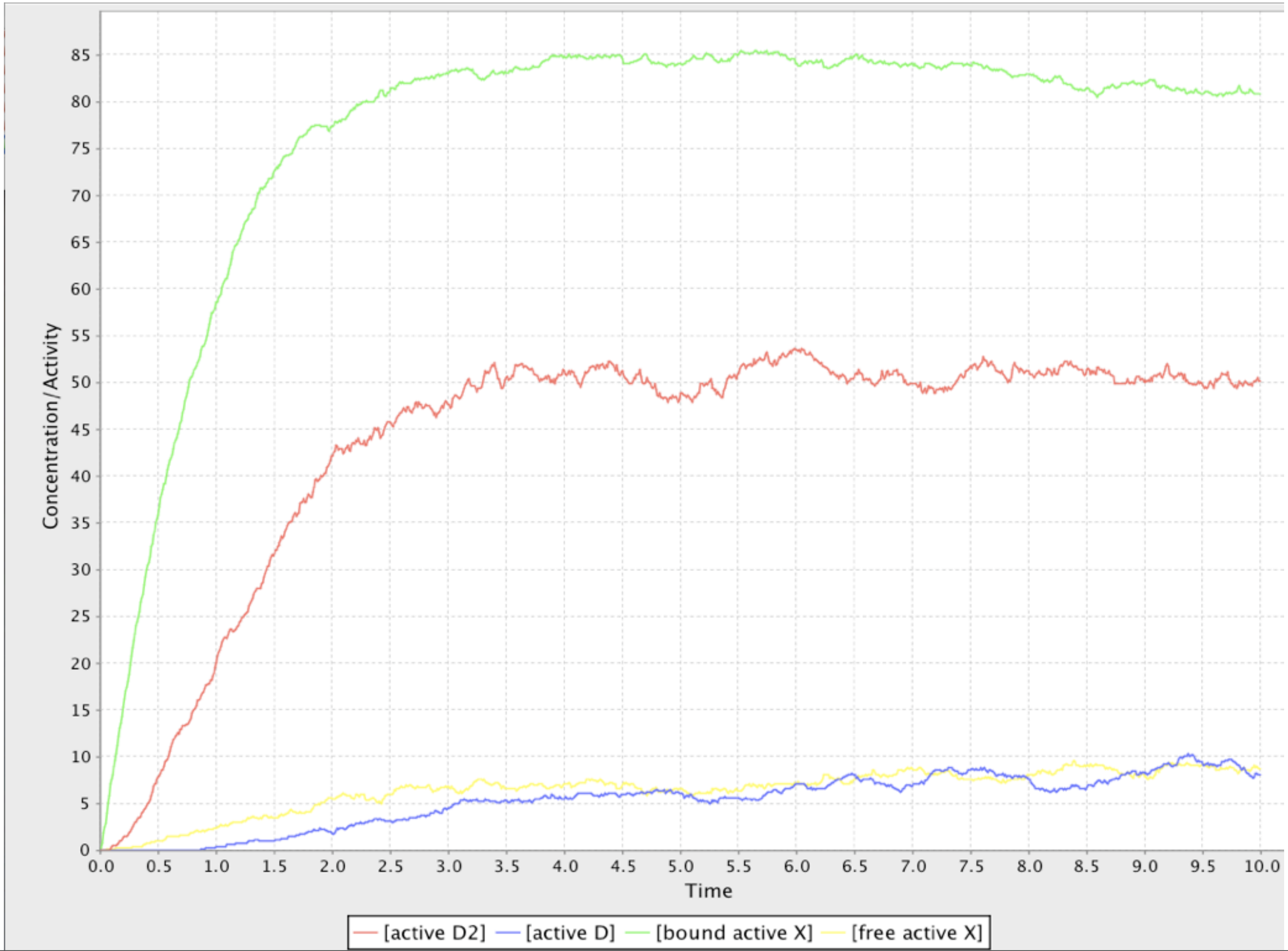
- So D now exists in two distinct forms
- X has two different "binding configurations" (which depend on whether it is bound or not)

U only



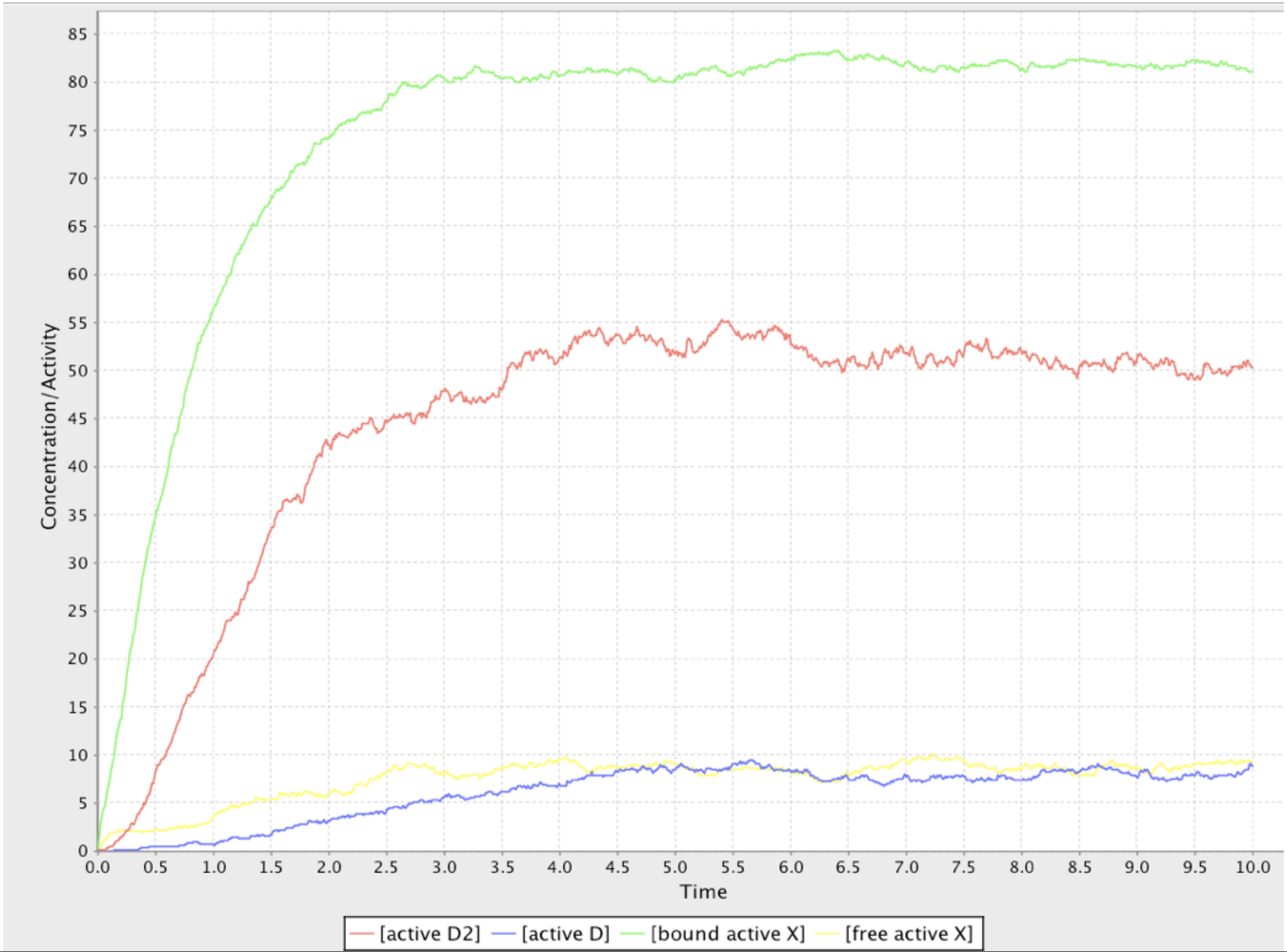
Concentration/Activity
Time
U
id
o
X
i
o
D
i
id
[active D2]
[active D]
[bound active X]
[free active X]

U2 only

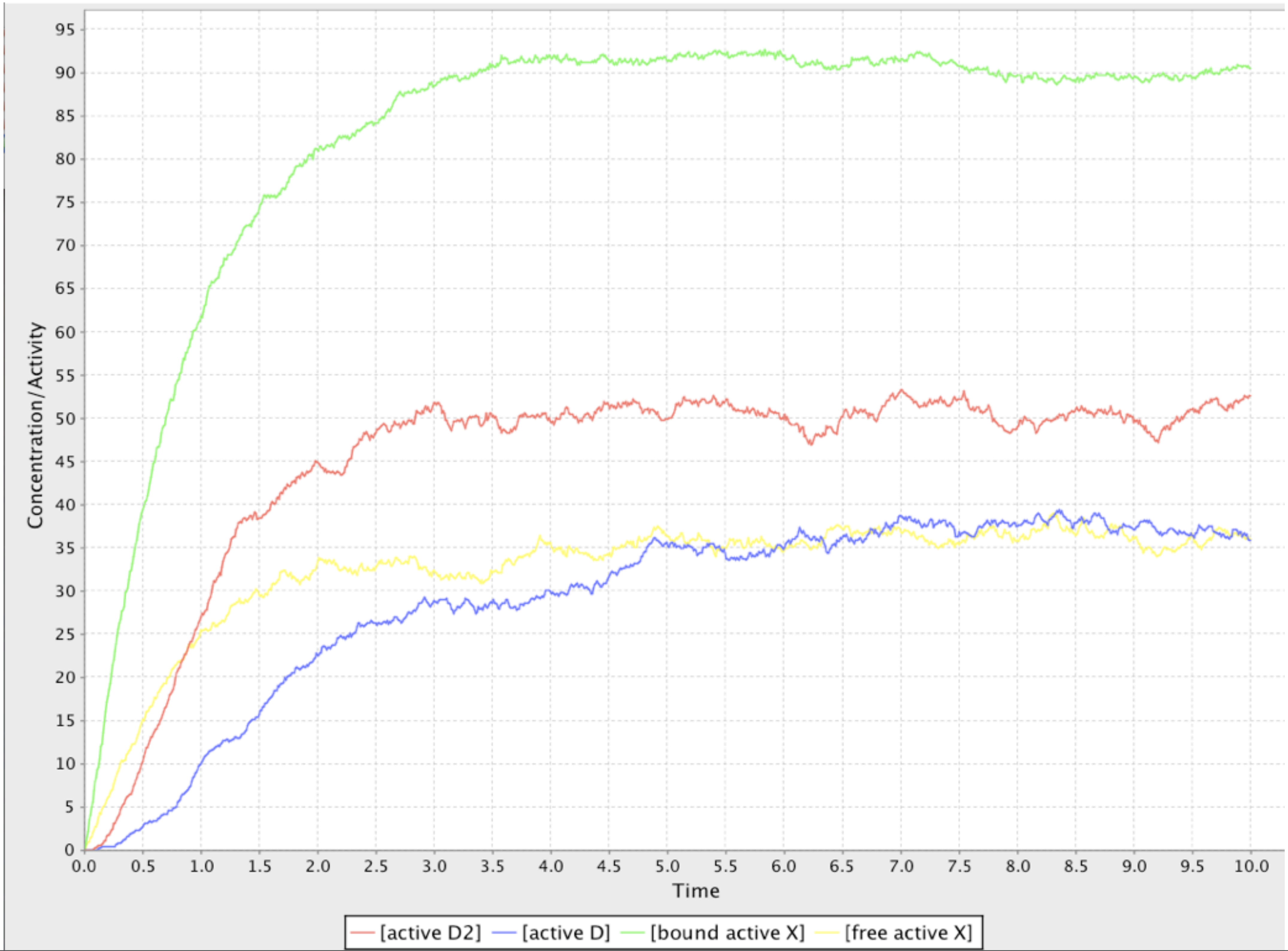


Concentration/Activity
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
Time
[active D2]
[active D]
[bound active X]
[free active X]

U and U2; X limited



U and U2; X in excess



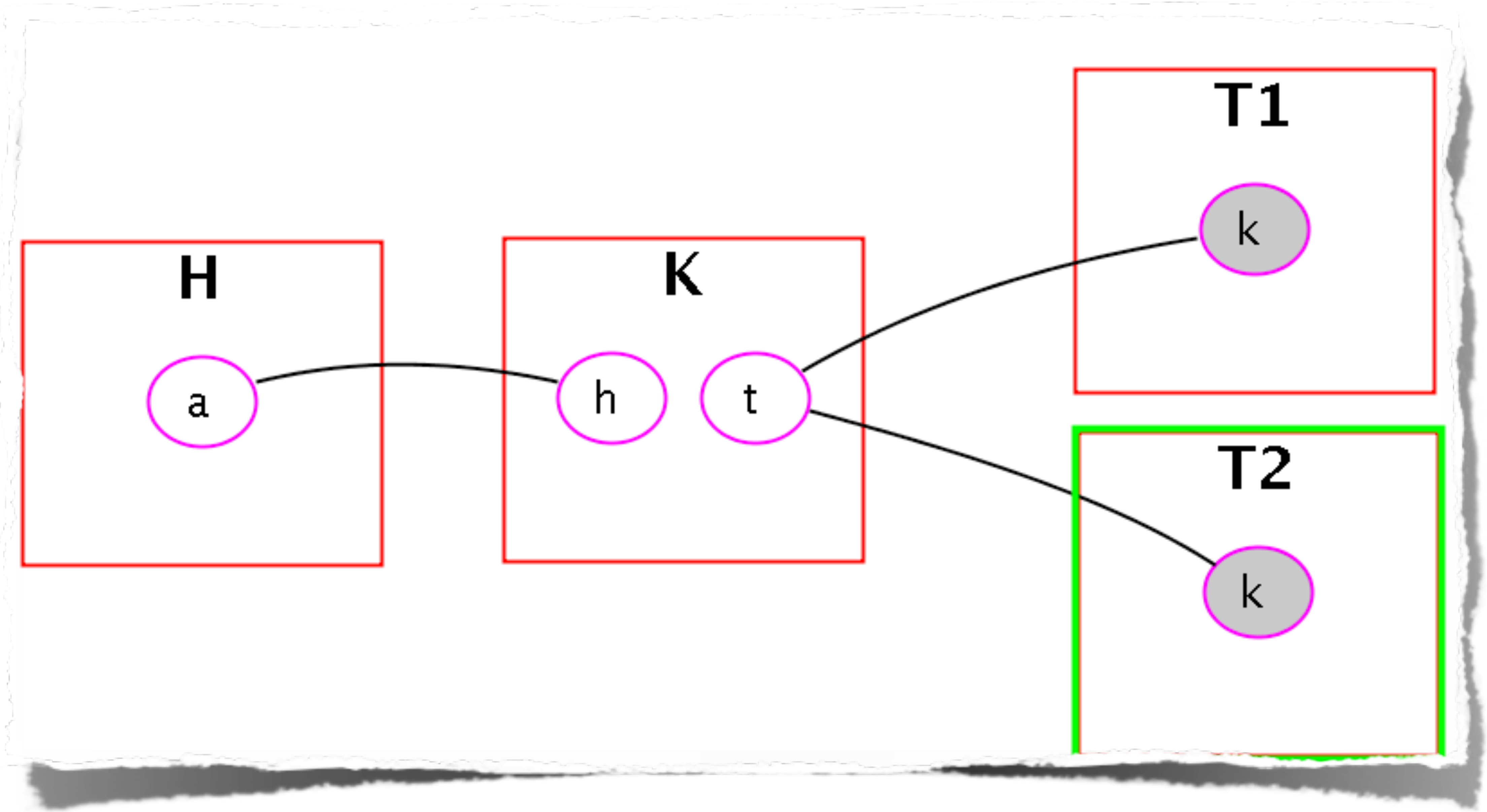
Specificity summary

- Little leakage (aka specificity):
  - `U` predominantly activates `D`
  - `U2` predominantly activates `D2`
- If `U` and `U2` are present,
  - limited `X` implies `U2` (high affinity) “wins”
  - excess `X` allows both `U` and `U2` to signal

Specificity (2)

- We might say that the `U2 -> X -> D2` pathway *hijacks* the original `U -> X -> D` pathway (when `X` is limited)
- Behaves analogously to a *transistor!*
- An example of *network plasticity:* the amount of `X` determines whether the network behaves as a transistor or simply as two parallel wires
  - a "scalpel" to divide a rule into pieces
  - a possible mechanism of "network evolution"

Another (puzzle)



players and a question

- `K -repairs> T1, T2` (as target)
- `K` also binds `H` (as helper)
- `T2` needs additional `H`
- how does `H` know where to go??
  - Swiss knife approach, saturate `K` with `H`
  - stochastic honey pot approach, uses refinements too

model 1: rules

- 'off KT1' K(t!1),T1(k!1) -> K(t),T1(k) @ 100
- 'on KT1' K(t),T1(k) -> K(t!1),T1(k!1) @ 1
- 'mod KT1' K(t!1),T1(k~no!1) -> K(t!1),T1(k~yes!1) @ 50

- 'unmod T1' T1(k~yes?) -> T1(k~no?) @ 1

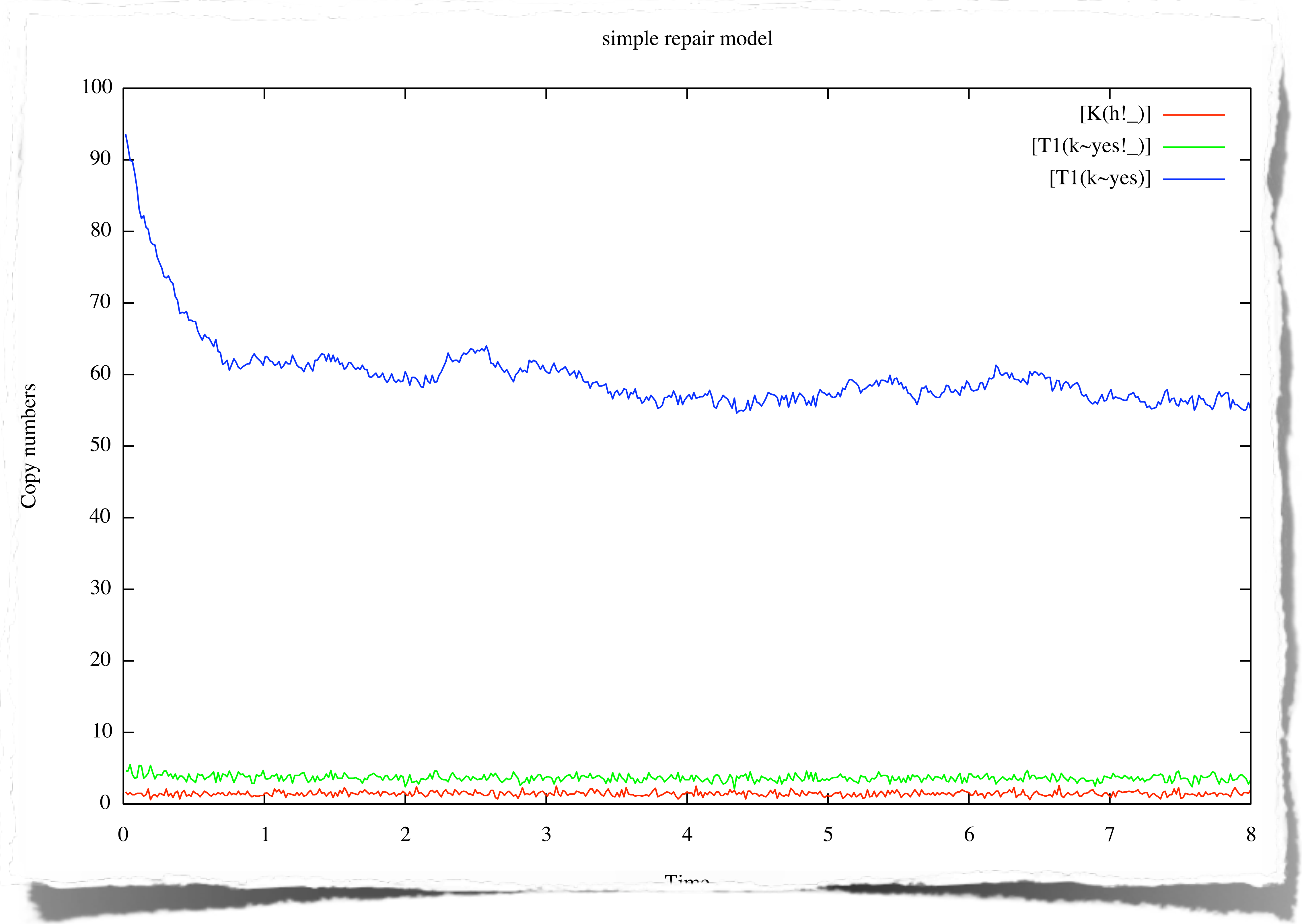
- 'off KH' K(h!1),H(k!1) -> K(h),H(k) @ 500
- 'on KH' K(h),H(k) -> K(h!1),H(k!1) @ 10

- %init: 10 * (K(h,t))
- %init: 10 * (H(k))
- %init: 100 * (T1(k~yes))

- %obs: T1(k~yes)
- %obs: T1(k~yes!_)
- %obs: K(h!_)

model 1: results

model 1: results



model 2: rules

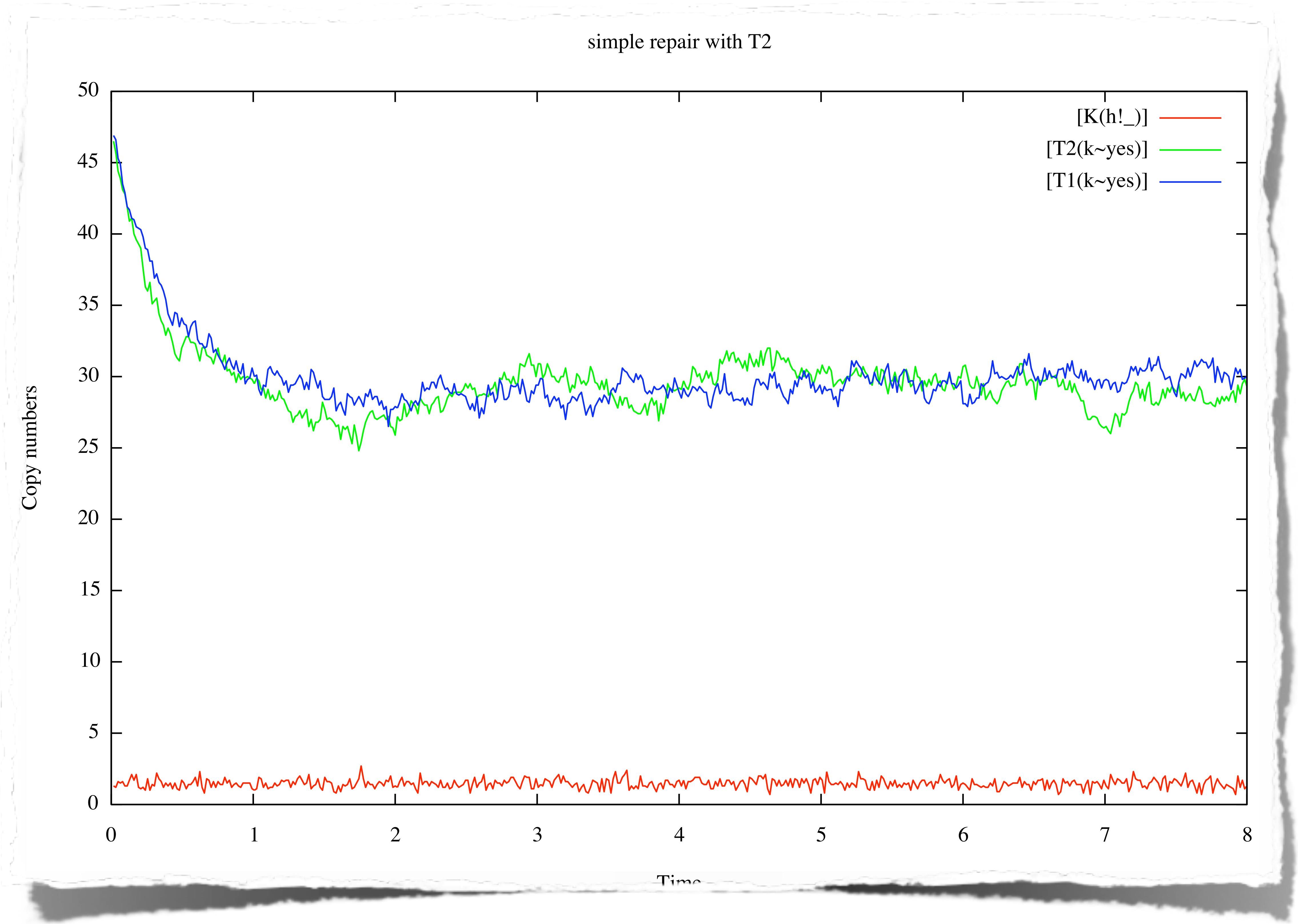
- 'off KT2' K(t!1),T2(k!1) -> K(t),T2(k) @ 100
- 'on KT2' K(t),T2(k) -> K(t!1),T2(k!1) @ 1
- 'mod KT2' K(t!1),T2(k~no!1) -> K(t!1),T2(k~yes!1) @ 50

- 'unmod T2' T2(k~yes?) -> T2(k~no?) @ 1

- %init: 10 * (K(h,t))
- %init: 10 * (H(k))
- %init: 50 * (T1(k~yes))
- %init: 50 * (T2(k~yes))

- %obs: T1(k~yes)
- %obs: T2(k~yes)
- %obs: K(h!_)

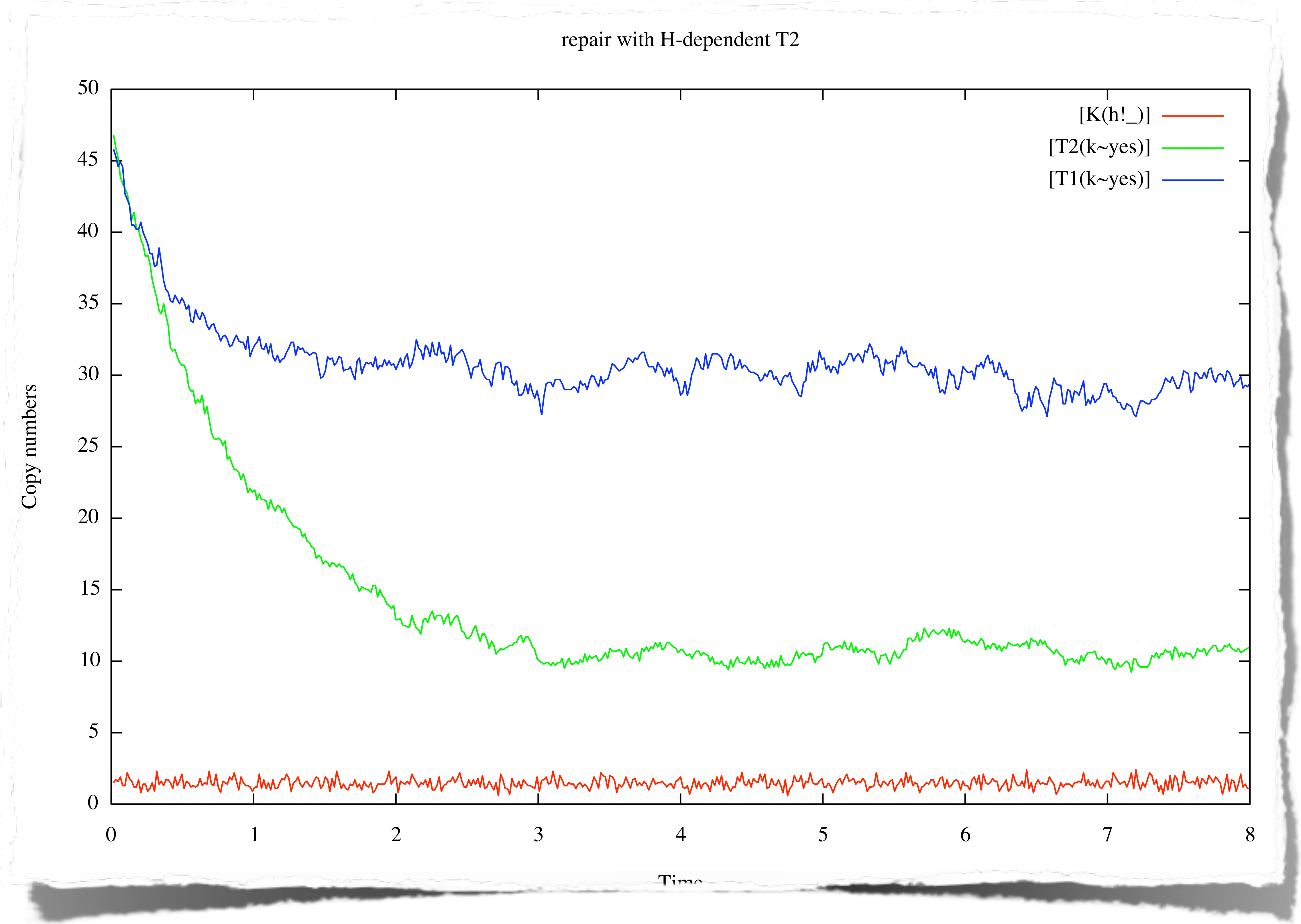
model 2: results



model 3: rules

- #'mod KT2' K(t!1),T2(k~no!1) -> K(t!1),T2(k~yes!1) @ 50
- 'mod KT2' K(t!1,h!_),T2(k~no!1) -> K(t!1,h!_),T2(k~yes!1) @ 50

model 3: results



model 4: rules

- #'off KH' K(h!1),H(k!1) -> K(h),H(k) @ 500
- K(h!1,t),H(k!1) -> K(h,t),H(k) @ 1000
- K(h!1,t!2),H(k!1),T1(k!2) -> K(h,t!2),H(k),T1(k!2) @ 1000
- K(h!1,t!2),H(k!1),T2(k!2) -> K(h,t!2),H(k),T2(k!2) @ 0.001

model 4: results