Debugging Maude Programs

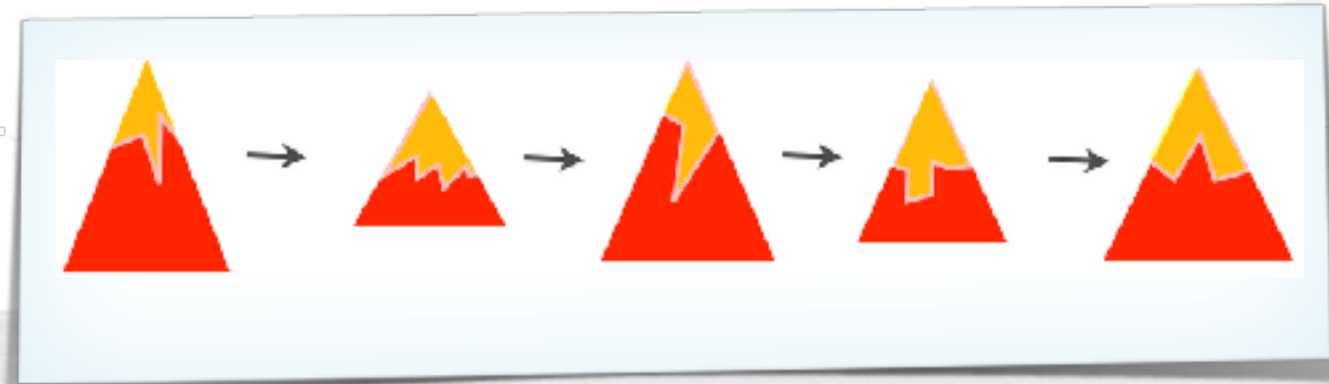
Demis Ballis

DMIF - University of Udine
email: demis.ballis@uniud.it

~~*Debugging Maude Programs*~~

Demis Ballis

DMIF - University of Udine
email: demis.ballis@uniud.it



Towards the Automated Debugging of Maude Programs

Demis Ballis

DMIF - University of Udine
email: demis.ballis@uniud.it

About this talk

* Techniques for debugging Maude programs with an increasing level of automation

* Joint work with great people at UPV
  * María Alpuente
  * Francisco Frechina
  * Daniel Romero
  * Julia Sapiña

Talk plan

* Rewriting logic and Maude (quick and dirty intro)
* Exploring Maude computations
* Debugging via backward trace slicing
* Debugging via automatic, assertion-based trace slicing
* Conclusion

Rewriting Logic

* Rewriting Logic (RWL) is a logical and semantic framework, which is particularly suitable for implementing and analyzing highly concurrent, complex systems
    * network protocols
    * biological systems
    * web apps
* RWL has been efficiently implemented in the programming language Maude.

RWL specifications

Equational Theory $\left\{ \right.$

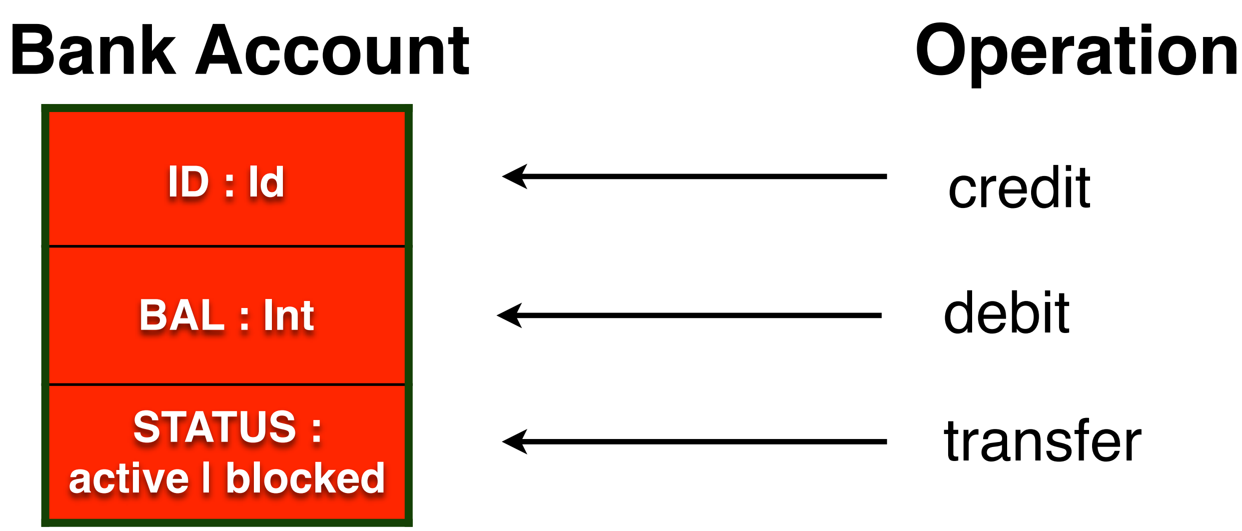
- A signature (i.e. set of operators) Σ
- A set of equations Δ
- A set of algebraic axioms (e.g. `comm`, `assoc`, `unity`) B

- A set of rewrite rules R

A RWL specification is a rewrite theory (i.e., a **Maude program**)

$$(\Sigma, \Delta \cup B, R)$$

A banking system



System State (Account | Msg)*

< Alice | 50 | active > ; < Bob | 40 | active > ; debit(Alice, 60)

A banking system

```
mod BANK is inc BANK-EQ .
    vars ID ID1 ID2 : Id .
    vars BAL BAL1 BAL2 M : Int .

    op empty-state : -> State [ctor] .
    op _;_ : State State -> State [ctor assoc comm id: empty-state] .
    ops credit debit : Id Int -> Msg [ctor] .
    op  transfer : Id Id Int -> Msg  [ctor] .

    rl [credit] : credit(ID,M) ; < ID | BAL | active >  => updSt(< ID | BAL + M | active >) .

    rl [debit] :  debit(ID,M) ; < ID | BAL | active >  => updSt(< ID | BAL - M | active >) .

    rl [transfer] : transfer(ID1,ID2,M) ; < ID1 | BAL1 | active > ; < ID2 | BAL2 | active >
                  => updSt(< ID1 | BAL1 - M | active >) ; updSt(< ID2 | BAL2 + M | active >) .

endm
```

A banking system

```
fmod BANK-EQ is inc BANK-INT+ID . pr SET{Id} .
    sorts Status Account PremiumAccount Msg State .
    subsort PremiumAccount < Account .
    subsorts  Account Msg < State .

    var ID : Id .                        op <_|_|_> : Id Int Status -> Account [ctor] .
    var BAL : Int .                  op active : -> Status [ctor] .
    var STS : Status .                   op blocked : -> Status [ctor] .

    op Alice : -> Id [ctor] .            op Bob : -> Id [ctor] .

    op PreferredClients : -> Set{Id} .
    eq PreferredClients = Bob .
   cmb < ID | BAL | STS > : PremiumAccount if ID in PreferredClients .

    op  secure : Account -> Account .

    ceq updSt(< ID | BAL | active >) = < ID | BAL | blocked > if BAL < 0 .
    eq  updSt(< ID | BAL | STS >) = < ID | BAL | STS > [owise] .

endfm
```

An active account
is blocked
if it is in the red

A banking system

```
fmod BANK-EQ is inc BANK-INT+ID . pr SET{Id} .
    sorts Status Account PremiumAccount Msg State .
    subsort PremiumAccount < Account .
    subsorts  Account Msg < State .

    var ID : Id .                    op <_|_|_> : Id Int Status -> Account [ctor] .
    var BAL : Int .                  op active : -> Status [ctor] .
    var STS : Status .               op blocked : -> Status [ctor] .

    op Alice : -> Id [ctor] .        op Bob : -> Id [ctor] .

    op PreferredClients : -> Set{Id} .
    eq PreferredClients = Bob .
    cmb < ID | BAL | STS > : PremiumAccount if ID in PreferredClients .

    op  secure : Account -> Account .
    ceq secure(< ID | BAL | active >) = < ID | BAL | blocked > if BAL <
    eq  secure(< ID | BAL | STS >) = < ID | BAL | STS > [owise] .
endfm
```

PreferredClients own PremiumAccounts (allowed to be in the red)

Rewriting modulo equational theories

- The evaluation mechanism is rewriting modulo equational theory ($\rightarrow_{R/\Delta\cup B}$)
  - Lifting the usual rewrite relation over terms to the congruence classes induced by the equational theory $(\Sigma, \Delta \cup B)$
  - Unfortunately, $\rightarrow_{R/\Delta\cup B}$ is in general undecidable since a rewrite step $t \rightarrow_{R/\Delta\cup B} t'$ involves searching through the possibly infinite equivalence classes of t and t'

Rewriting modulo equational theories

- Maude implements $\to_{R/\Delta\cup B}$ using two much simpler rewrite relations $\to_{\Delta,B}$ and $\to_{R,B}$ that use an algorithm of matching modulo B
  - $\to_{\Delta,B}$ rewrites terms using equations/axioms as simplification rules
  - For any term t, by repeatedly applying the equations/axioms, we eventually reach a canonical form $t{\downarrow_\Delta}$ to which no further equations can be applied

must be Church-Rosser and terminating!

Rewriting modulo equations and axioms

- Maude implements $\to_{R/\Delta\cup B}$ using two much simpler rewrite relations $\to_{\Delta,B}$ and $\to_{R,B}$ that use an algorithm of matching modulo B
  - $\to_{\Delta,B}$ rewrites terms using equations in Δ as simplification rules
  - $\to_{R,B}$ rewrites terms using rewrite rules in R

Rewrite steps

- a rewrite step modulo $\Delta \cup B$ on a term t can be implemented by applying the following rewrite strategy:

  1. reduce t w.r.t. $\rightarrow_{\Delta,B}$ until the canonical form $t{\downarrow_\Delta}$ is reached;

  2. rewrite $t{\downarrow_\Delta}$ w.r.t. $\rightarrow_{R,B}$ to t'.

$$\boxed{t \rightarrow^{*}_{\Delta,B} t{\downarrow_\Delta} \rightarrow_{R,B} t'}$$

RWL traces

* A trace (computation) in the rewrite theory $(\Sigma, \Delta \cup B, R)$ is a (possibly infinite) rewrite sequence of the form:

$$s_0 \rightarrow^*_{\Delta,B} s_0{\downarrow_\Delta} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} s_1{\downarrow_\Delta} \ldots$$

that interleaves rewrite steps with equations and rules following the reduction strategy previously mentioned.

* the terms that appear in a computation are also called states.

RWL traces: example

< Alice | 50 | active > ; < Bob | 40 | active > ; debit(Alice, 30)

```
rl [debit] :  debit(ID,M) ; < ID | BAL | active >  => updSt(< ID | BAL - M | active >) .
```

RWL traces: example

< Alice | 50 | active > ; < Bob | 40 | active > ; debit(Alice, 30)

eq. simplification

debit(Alice, 30) ; < Alice | 50 | active > ; < Bob | 40 | active > ;

```
rl [debit] :  debit(ID,M) ; < ID | BAL | active >  => updSt(< ID | BAL - M | active >) .
```

RWL traces: example

< Alice | 50 | active > ; < Bob | 40 | active > ; debit(Alice, 30)

↓ eq. simplification

debit(Alice, 30) ; < Alice | 50 | active > ; < Bob | 40 | active > ;

↓ debit application

UpdSt(< Alice | 50 - 30 | active >) ; < Bob | 40 | active >

```
rl [debit] :  debit(ID,M) ; < ID | BAL | active >  => updSt(< ID | BAL - M | active >) .
```

RWL traces: example

< Alice | 50 | active > ; < Bob | 40 | active > ; debit(Alice, 30)

↓ eq. simplification

debit(Alice, 30) ; < Alice | 50 | active > ; < Bob | 40 | active > ;

↓ debit application

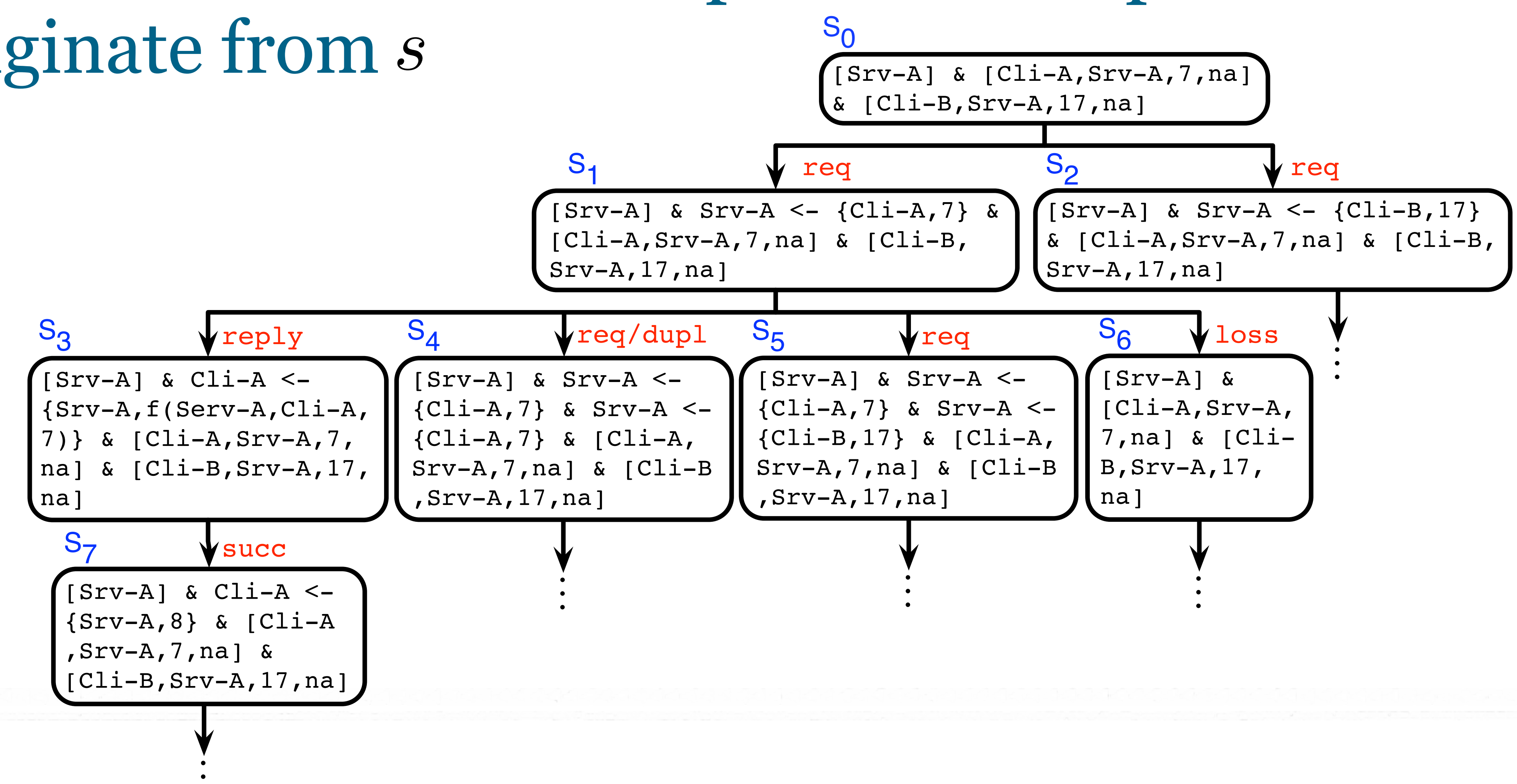
UpdSt(< Alice | 50 - 30 | active >) ; < Bob | 40 | active >

↓* eq. simplification

< Alice | 20 | active > ; < Bob | 40 | active >

Computation trees

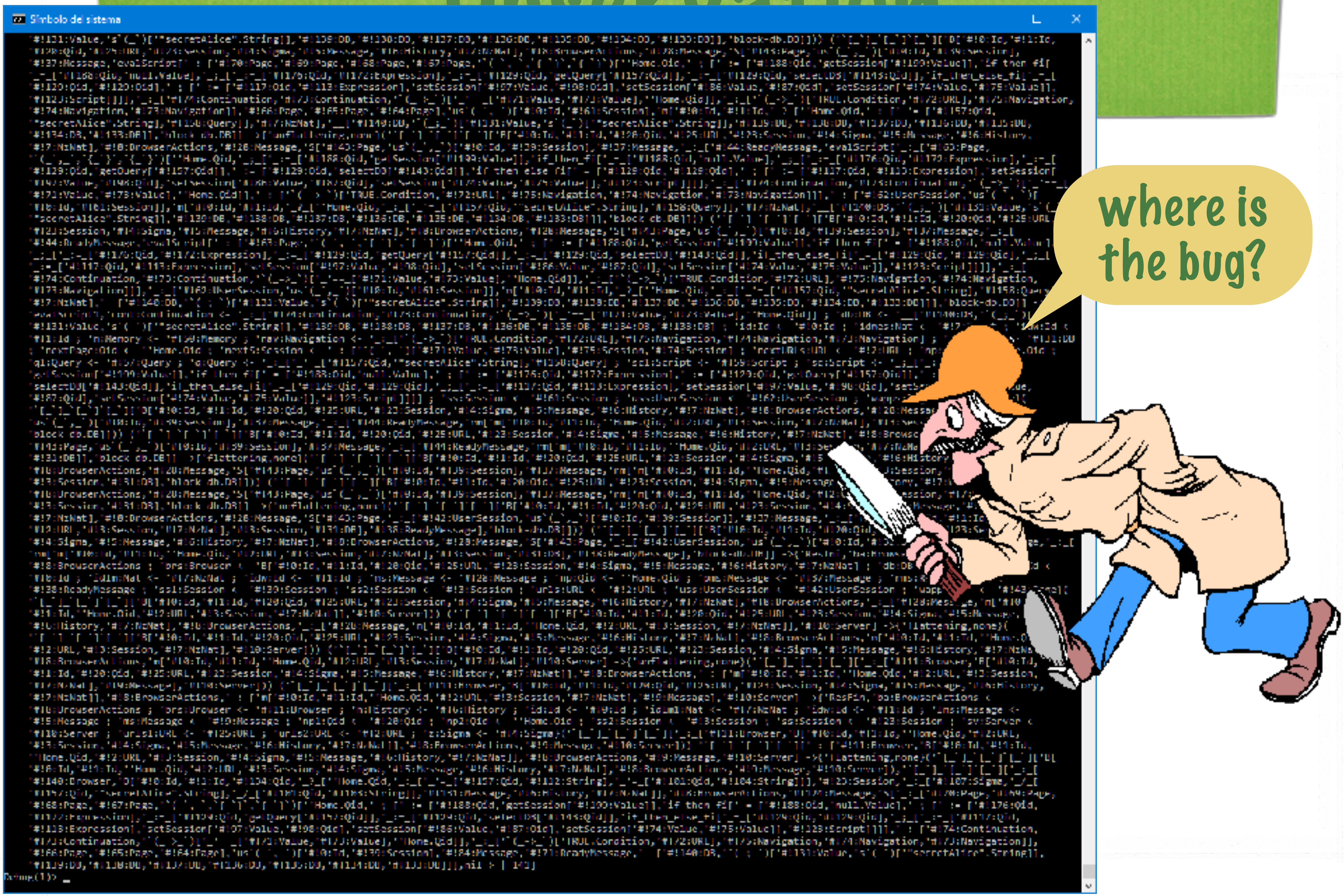
- Given a rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, a computation tree $\mathcal{T}_{\mathcal{R}}(s)$ for a term s is a tree-like representation of all the possible computations that originate from s



Observation

- Computation trees are typically large (possibly infinite) and complex objects to deal with because of the highly-concurrent, non-deterministic nature of Rewriting Logic theories.
- Inspecting computation trees using the Maude built-in program tracer could be painful
    - textual output
    - implicit axiom applications

Observation



Exploring computations

* Computations can be manually explored to detect program misbehaviours
* To facilitate exploration...
  * use a graphical representation of the computation tree
  * define a stepwise, user-driven, computation exploration technique

Exploring computations

* Computations can be manually explored to detect program misbehaviours
* To facilitate exploration...
  * use a graphical representation of the computation tree
  * define a stepwise, user-driven, computation exploration technique

➔ **the ANIMA tool**

ANIMA

* ANIMA is a visual program animator for Maude
* Available as a web service at:

  `http://safe-tools.dsic.upv.es/anima/`

* States in a computation can be expanded/folded by a simple "point and click" strategy

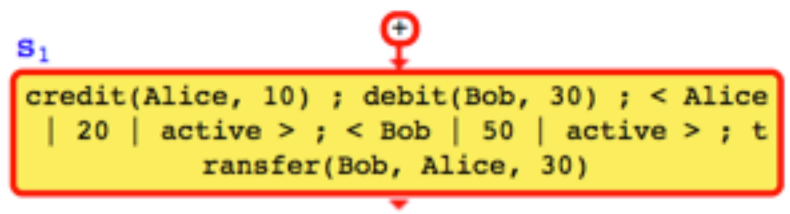
ANIMA

* ANIMA is a visual program animator for Maude
* Available as a web service at:

  http://safe-tools.dsic.upv.es/anima/

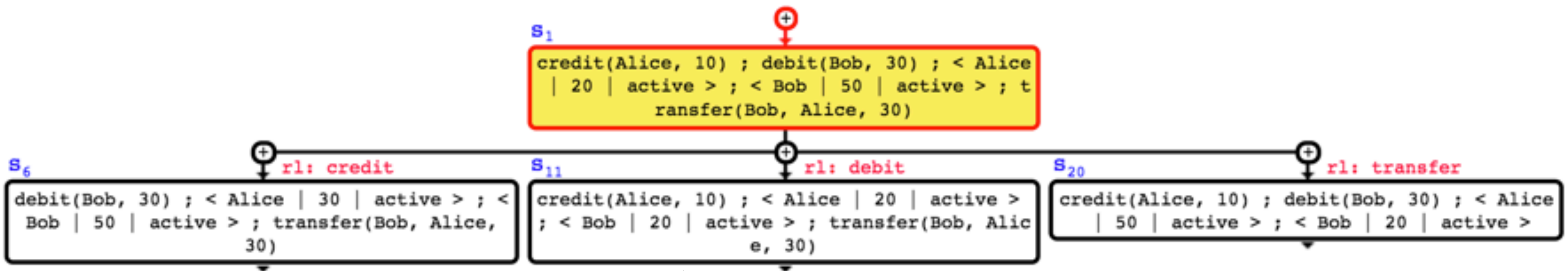
* States in a computation can be expanded/folded by a simple “point and click” strategy

ANIMA



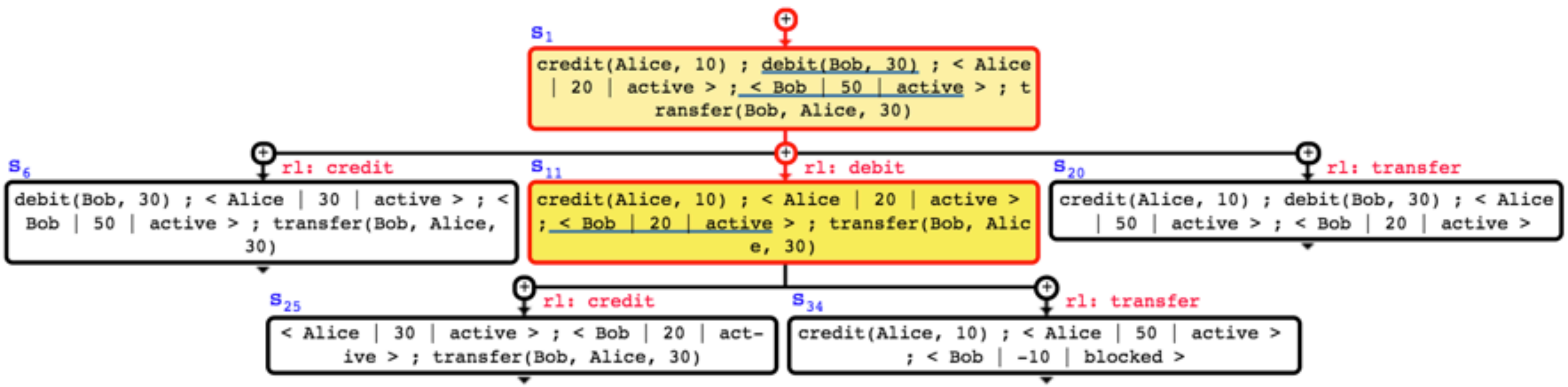
Bob's account active
current balance 50 euros

ANIMA



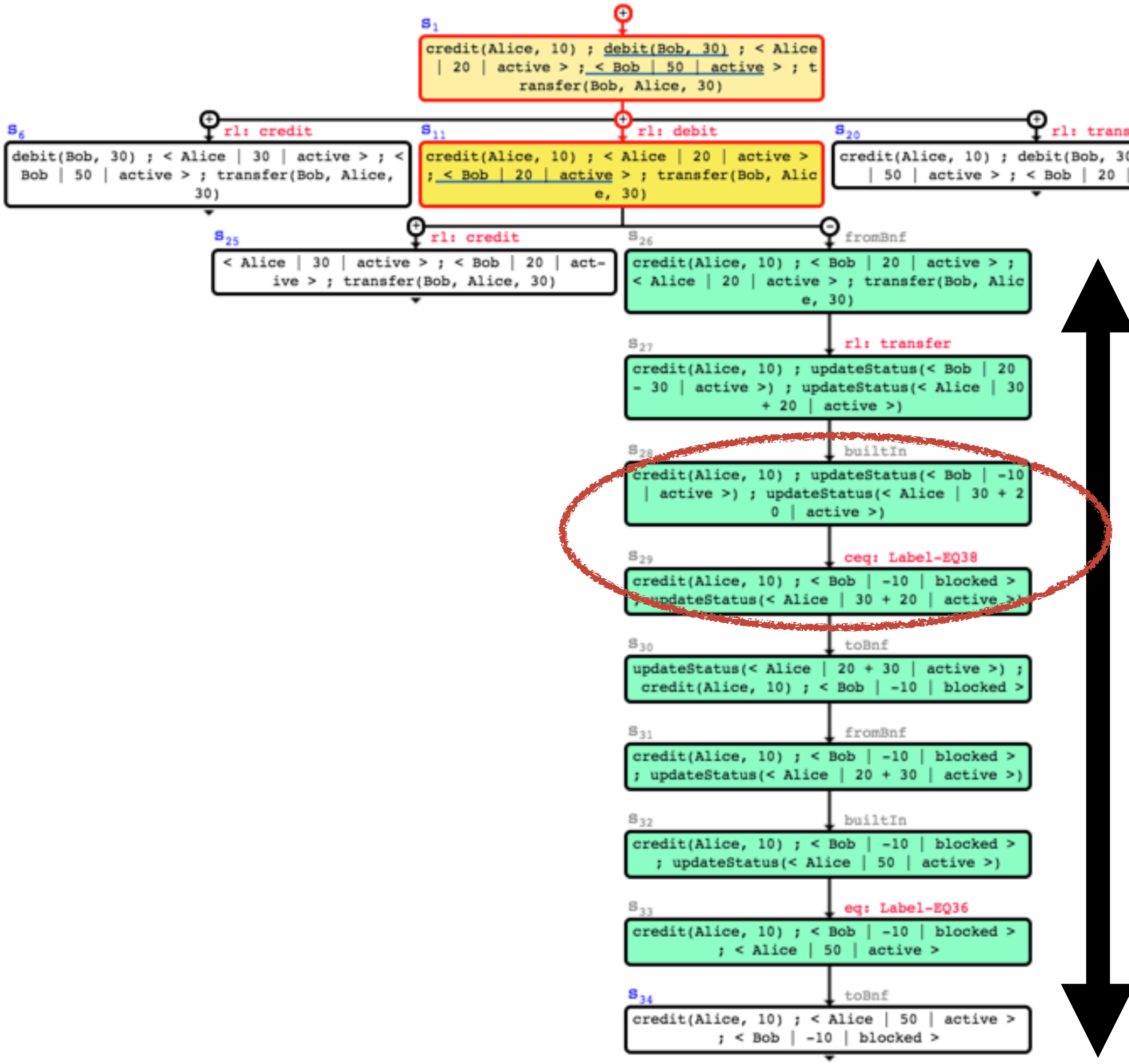
Bob's account active
current balance 20 euros

ANIMA



Bob's account blocked
current balance -10 euros

Error: Bob is a premium client and premium accounts cannot be blocked



Equational simplification that leads to the wrong state

updateStatus function call erroneously blocks bob's account

Some remarks

* Debugging via program animation
    * it’s ok for simple programs
    * completely manual
    * navigation through quite a lot of information

Some remarks

- Debugging via program animation
    - it's ok for simple programs
    - completely manual
    - navigation through quite a lot of information

Question: can we somehow reduce the size of the computations to favor their inspection?

Yes, we can: trace slicing

- Trace slicing is a transformation technique that reduces the complexity of execution trace

- Based on tracking origins/descendants

- It favors better analysis and debugging since irrelevant inspections can be eliminated automatically

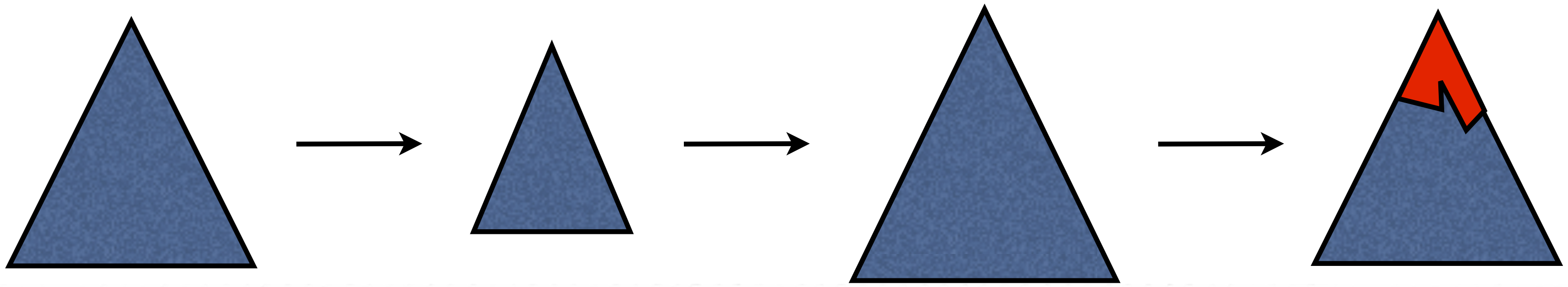
Backward trace slicing

Definition (Backward Trace Slicing)

Given an execution trace $\mathcal{T}$ and a slicing criterion O for the trace (i.e., data we want to observe in the final state of the trace),

- traverse $\mathcal{T}$ from back to front, and at each rewrite step,
- incrementally compute the origins of the observed data
- remove the irrelevant data

Trace



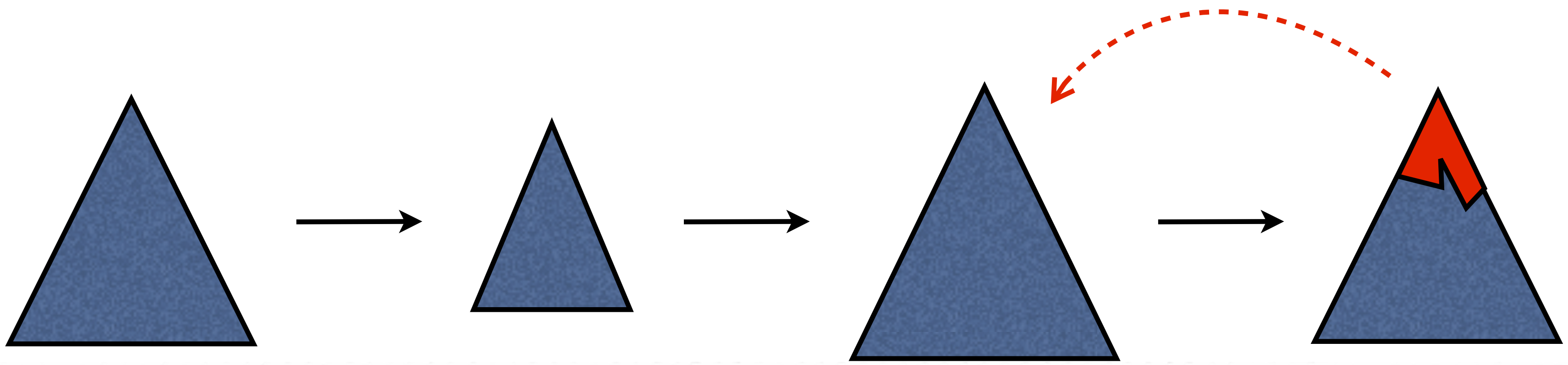
Backward trace slicing

Definition (Backward Trace Slicing)

Given an execution trace $\mathcal{T}$ and a slicing criterion O for the trace (i.e., data we want to observe in the final state of the trace),

- traverse $\mathcal{T}$ from back to front, and at each rewrite step,
- incrementally compute the origins of the observed data
- remove the irrelevant data

Trace



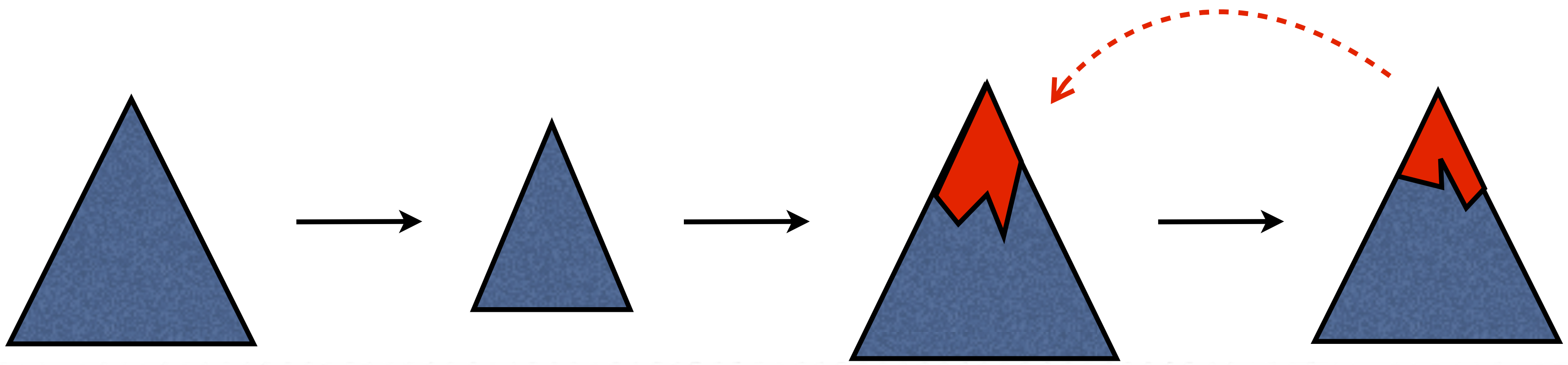
Backward trace slicing

Definition (Backward Trace Slicing)

Given an execution trace $\mathcal{T}$ and a slicing criterion O for the trace (i.e., data we want to observe in the final state of the trace),

- traverse $\mathcal{T}$ from back to front, and at each rewrite step,
- incrementally compute the origins of the observed data
- remove the irrelevant data

Trace



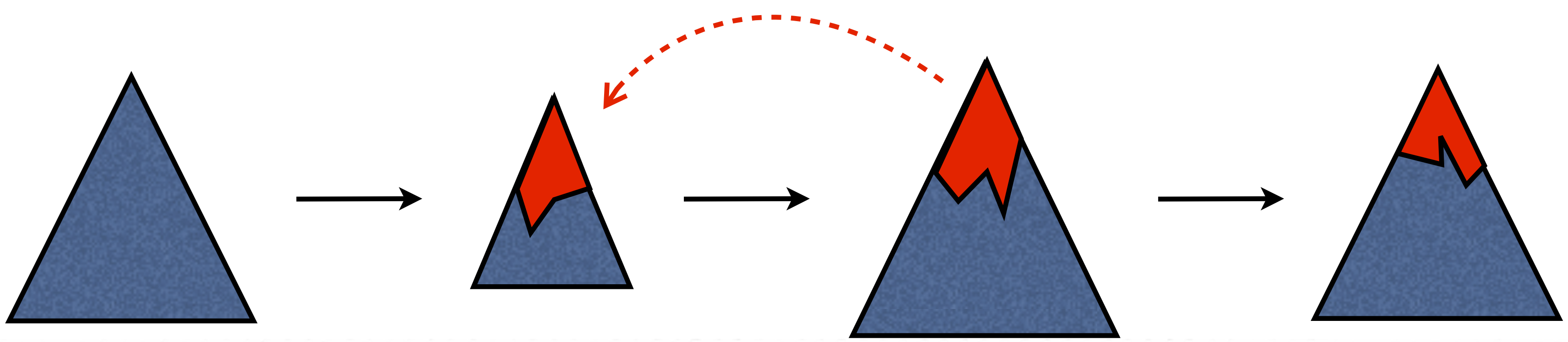
Backward trace slicing

Definition (Backward Trace Slicing)

Given an execution trace $\mathcal{T}$ and a slicing criterion O for the trace (i.e., data we want to observe in the final state of the trace),

- traverse $\mathcal{T}$ from back to front, and at each rewrite step,
- incrementally compute the origins of the observed data
- remove the irrelevant data

Trace



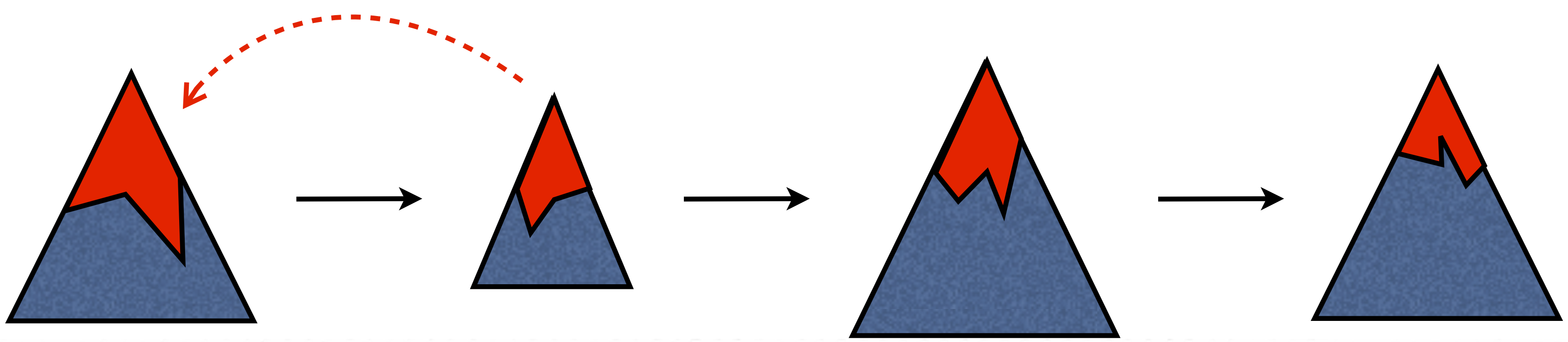
Backward trace slicing

Definition (Backward Trace Slicing)

Given an execution trace $\mathcal{T}$ and a slicing criterion O for the trace (i.e., data we want to observe in the final state of the trace),

- traverse $\mathcal{T}$ from back to front, and at each rewrite step,
- incrementally compute the origins of the observed data
- remove the irrelevant data

Trace



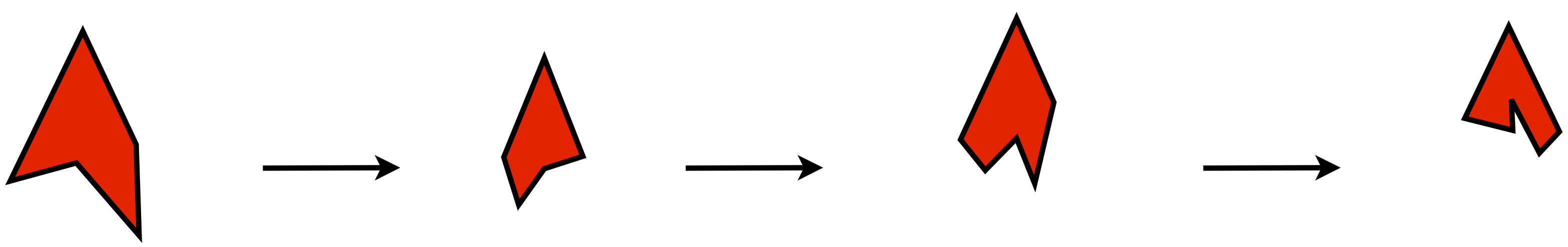
Backward trace slicing

Definition (Backward Trace Slicing)

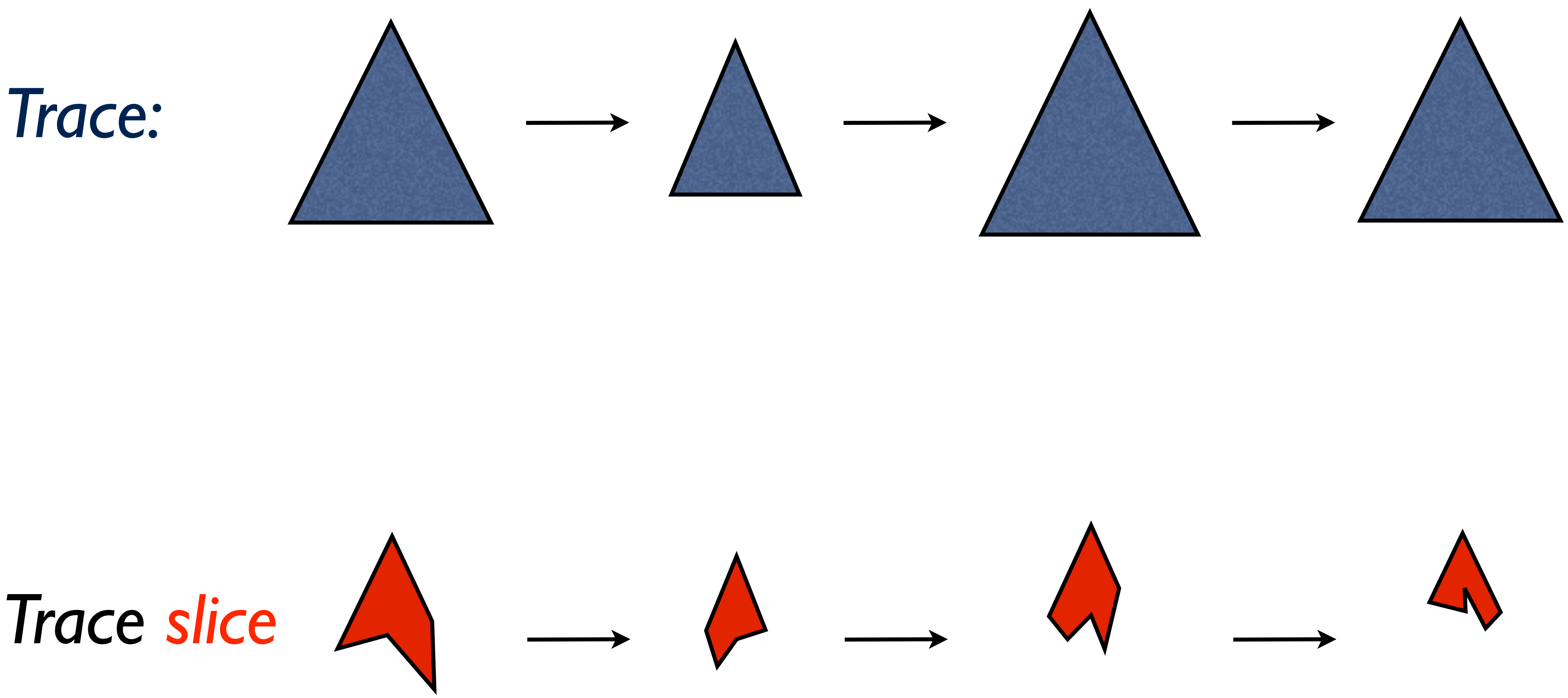
Given an execution trace $\mathcal{T}$ and a slicing criterion O for the trace (i.e., data we want to observe in the final state of the trace),

- traverse $\mathcal{T}$ from back to front, and at each rewrite step,
- incrementally compute the origins of the observed data
- remove the irrelevant data

Trace slice



Backward Trace slicing



iJulienne

* iJulienne is a backward trace slicer for Maude
* Available as a web service at:

`http://safe-tools.dsic.upv.es/ijulienne/`

iJulienne: exa

Note: Alice's account balance is negative and she is a regular client

* Let us feed iJulienne with the computation

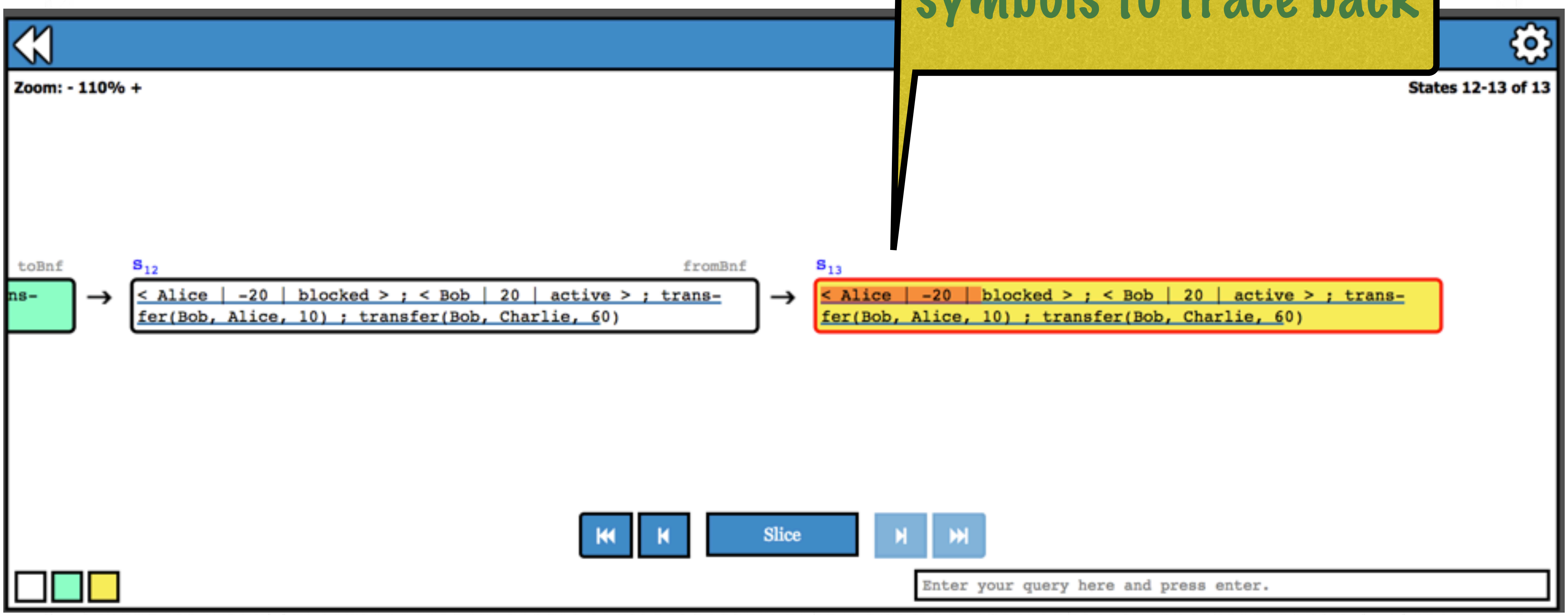
```
< Alice | 50 | active > ; < Bob | 20 | active > ;
debit(Alice, 30) ; transfer(Bob, Charlie, 60) ;
debit(Alice, 40) ; transfer(Bob, Alice, 10)
```

$$\downarrow^{*}$$

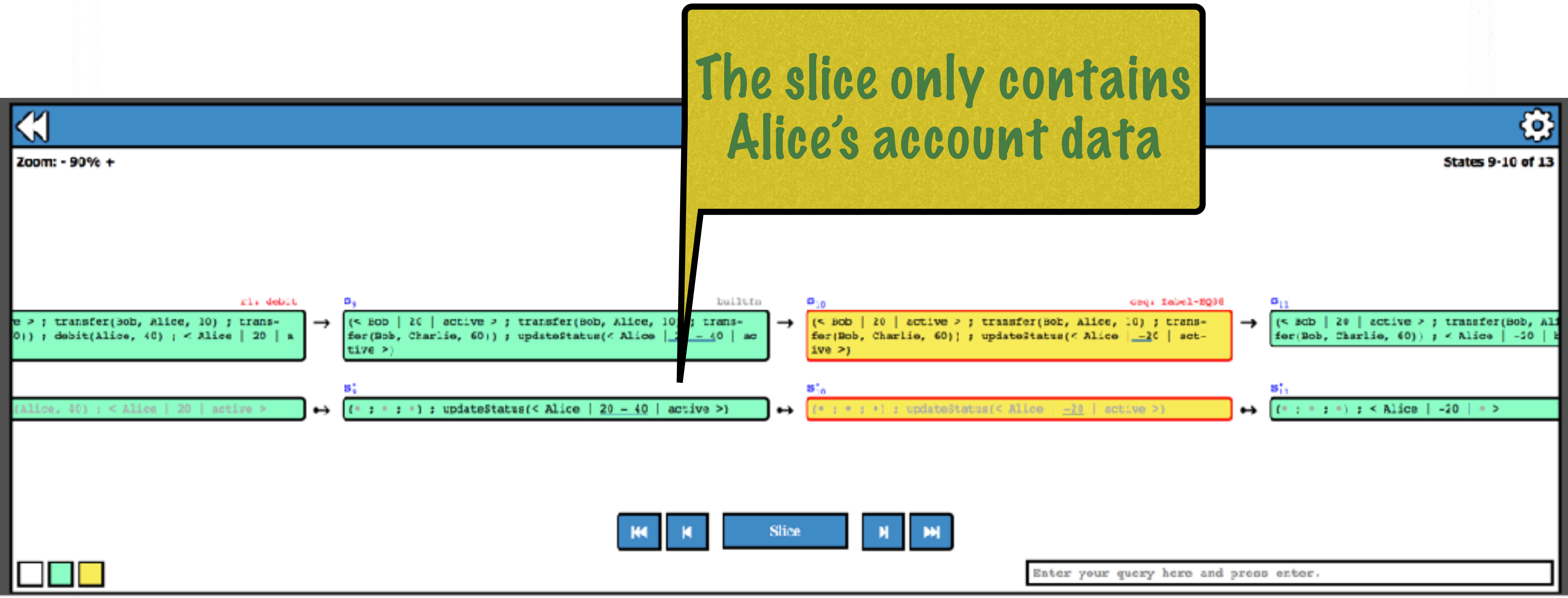
```
< Alice | -20 | blocked > ; < Bob | 20 | active > ;
transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)
```

iJulienne: example

Select the suspicious symbols to trace back



iJulienne: example



iJulienne: example

Trace information (trusted mode)

State	Label	Original trace	Sliced trace
1	'Start	< Alice | 50 | active > ; < Bob | 20 | active > ; debit(Alice, 30) ; transfer(Bob, Charlie, 60) ; debit(Alice, 40) ; transfer(Bob, Alice, 10)	**< Alice | 50 | active > ; • ; debit(Alice, 30) ; • ; debit(Alice, 40) ; •**
2	toBnf	debit(Alice, 30) ; debit(Alice, 40) ; < Alice | 50 | active > ; < Bob | 20 | active > ; transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)	debit(Alice, 30) ; debit(Alice, 40) ; < Alice | 50 | active > ; • ; • ; •
		…ive > ; transfer(Bob, Alice, 1 … ebit(Alice, 30) ; < Alice | 50	(debit(Alice, 40) ; • ; • ; •) ; debit(Alice, 30) ; < Alice | 50 | active >
		…ve > ; transfer(Bob, Alice, 1 … pdateStatus(< Alice | 50 - 30	**(debit(Alice, 40) ; • ; • ; •) ; updateStatus(< Alice | 50 - 30 | active >)**
		…ve > ; transfer(Bob, Alice, 1 … pdateStatus(< Alice | 20 | ac	(debit(Alice, 40) ; • ; • ; •) ; updateStatus(< Alice | 20 | active >)
6	Label-EQ36	(debit(Alice, 40) ; < Bob | 20 | active > ; transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)) ; < Alice | 20 | active >	**(debit(Alice, 40) ; • ; • ; •) ; < Alice | 20 | active >**
7	toBnf	debit(Alice, 40) ; < Alice | 20 | active > ; < Bob | 20 | active > ; transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)	debit(Alice, 40) ; < Alice | 20 | active > ; • ; • ; •
8	fromBnf	(< Bob | 20 | active > ; transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)) ; debit(Alice, 40) ; < Alice | 20 | active >	(• ; • ; •) ; debit(Alice, 40) ; < Alice | 20 | active >
9	debit	(< Bob | 20 | active > ; transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)) ; updateStatus(< Alice | 20 - 40 | active >)	**(• ; • ; •) ; updateStatus(< Alice | 20 - 40 | active >)**
10	builtIn	(< Bob | 20 | active > ; transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)) ; updateStatus(< Alice | -20 | active >)	(• ; • ; •) ; updateStatus(< Alice | -20 | active >)
11	Label-EQ38	(< Bob | 20 | active > ; transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)) ; < Alice | -20 | blocked >	**(• ; • ; •) ; < Alice | -20 | • >**
12	toBnf	< Alice | -20 | blocked > ; < Bob | 20 | active > ; transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)	< Alice | -20 | • > ; • ; • ; •
13	fromBnf	< Alice | -20 | blocked > ; < Bob | 20 | active > ; transfer(Bob, Alice, 10) ; transfer(Bob, Charlie, 60)	< Alice | -20 | • > ; •

Bad implementation of the debit rule!

Some remarks

Debugging via Backward trace slicing allows the information to be inspected to be (greatly) reduce

but...

Some remarks

Debugging via Backward trace slicing allows the information to be inspected to be (greatly) reduce

but...

It requires the user to manually select the slicing criterion (i.e. the data to be observed)

⬇

It cannot be used to fully automatize debugging

Assertion-based backward trace slicing

Backward trace slicing

coupled with

Assertion checking

Assertion-based slicing technique that

- automatically infers the slicing criterion
- and use it to automatically fire the slicer

Assertion language

We define assertions by using ***constrained terms*** S{ϕ}, where

- S is a non-ground term *(state template)*
- **ϕ** is a quantifier-free boolean formula

ϕ = **true** | **false** | p(t1,…,tn) | ϕ **and** ϕ | **not** ϕ | ϕ **implies** ϕ

Assertion language

We define two groups of assertions:

- **System assertions** $\boxed{S\{\phi\}}$

  $Var(\phi) \subseteq Var(S)$

invariant properties of the system states *t* *"No employee is under age 18"*

- **Functional assertions** $\boxed{I\{\phi_{in}\} \rightarrow O\{\phi_{out}\}}$

  $Var(\phi_{in}) \subseteq Var(I)$

  $Var(\phi_{out}) \subseteq Var(I) \cup Var(O)$

pre/post-conditions over equational simplification traces μ: $t \rightarrow^* t{\downarrow}_{\Delta,B}$ *"Sorting a list preserves its length"*

System assertion

“The account of a regular client can’t have a negative balance”

Θ = < C:Id | B:Int | S:Status >

{ not (C : Id in PreferredClients) implies B : Int >= 0 }

Then, Θ is satisfied in the state

< Alice | 50 | active > ; < Bob | 40 | active > ; debit(Alice, 60)

but it is not satisfied in

< Alice | -10 | blocked > ; < Bob | 40 | active >

Satisfaction for system assertions

$$t \models S\{\phi\}$$

iff

for each position w of t and substitution σ

$t|_w =_E S\sigma \Rightarrow \phi\sigma$ holds in the theory R

Satisfaction for system assertions

$$t \models S\{\phi\}$$

iff

for each position w of t and substitution σ

By-product:

System error symptom ξ_{sys}

$$\boxed{t|_w =_E S\sigma} \Rightarrow \phi\sigma \text{ ~~holds in~~ } R$$

ξ_{sys}: Any position w pointing to a subterm of t that

- equationally matches the template S, but
- the E-matcher produces a false instance of the formula ϕ

<Alice | -10 | blocked>

-10 >= 0

Satisfaction for system assertions

< Alice | -10 | blocked > ; < Bob | 40 | active >

↓

Slicing criterion   < Alice | -10 | blocked > ; ●

Satisfaction for functional assertion

Given $\mu = t \longrightarrow^{*}_{\Delta,B} t\downarrow_{\Delta,B}$

$$\boxed{\mu \models \mathrm{I}\ \{\phi_{in}\} \rightarrow \mathrm{O}\ \{\phi_{out}\}}$$

iff

for every substitution σ s.t.

$t =_B \mathrm{I}\sigma$ and $\phi_{in}\sigma$ holds in R,

there exists σ' s.t.

$t\downarrow_{\Delta,B} =_B \mathrm{O}(\sigma\downarrow_{\Delta,B})\sigma'$ and $\phi_{out}(\sigma\downarrow_{\Delta,B})\sigma'$ holds in R

Satisfaction for functional assertion

Given $\mu = t \longrightarrow^{*}_{\Delta,B} t{\downarrow_{\Delta,B}}$

$$\boxed{\mu \models I\,\{\phi_{in}\} \rightarrow O\,\{\phi_{out}\}}$$

iff

for every substitution σ s.t.

$$t =_B I\sigma \text{ and } \phi_{in}\sigma \text{ holds in } R,$$

there exists σ' s.t.

$$t{\downarrow_{\Delta,B}} \cancel{=_B} O(\sigma{\downarrow_{\Delta,B}})\sigma' \text{ or } \phi_{out}(\sigma{\downarrow_{\Delta,B}})\sigma' \text{ ~~holds~~ in } R$$

$t{\downarrow_{\Delta,B}}$ fails to fit the template O

σ' fails to verify the postcondition ϕ_{out}

Satisfaction for functional assertion

$t\downarrow_{\Delta,B}$ fails to fit the template O

σ' fails to verify the postcondition ϕ_{out}

$t\downarrow_{\Delta,B} \cancel{=_B}\ O(\sigma\downarrow_{\Delta,B})\sigma'$ or $\phi_{out}(\sigma\downarrow_{\Delta,B})\sigma'$ ~~holds~~ in R

By-product:
Functional error symptom ξ_{fun}

ξ_{fun}: Position in $t\downarrow_{\Delta,B}$ that disagrees with O or the pointed subterm causes σ' ***to*** falsify the postcondition ϕ_{out}

Satisfaction for functional assertion

"updSt is the identity function on premium accounts"

Φ = updSt(acc:Account) { isPremium(acc:Account) }
→ acc:Account { true }

Φ is not satisfied in this equational simplification

updSt(< Bob | -5 | active >) $\rightarrow^{+}$ < Bob | -5 | blocked >

Disagreement at position 3 of the wrong $t{\downarrow}_{\Delta,B}$ ✗

Satisfaction for functional assertion

Disagreements are computed via a least-general generalization algorithm modulo the equational theory E (antiunification modulo E)

$$\boxed{\mathrm{updSt}(<\mathrm{Bob} \mid -5 \mid \mathrm{active}>) \to^{+} <\mathrm{Bob} \mid -5 \mid \mathrm{blocked}>}$$

clash

$$<\mathrm{Bob} \mid -5 \mid \mathbf{x}>$$

Satisfaction for functional assertion

Disagreements are computed via a least-general generalization algorithm modulo the equational theory E (antiunification modulo E)

updSt(< Bob | -5 | active >) →⁺ < Bob | -5 | blocked >

clash

< Bob | -5 | **x** >

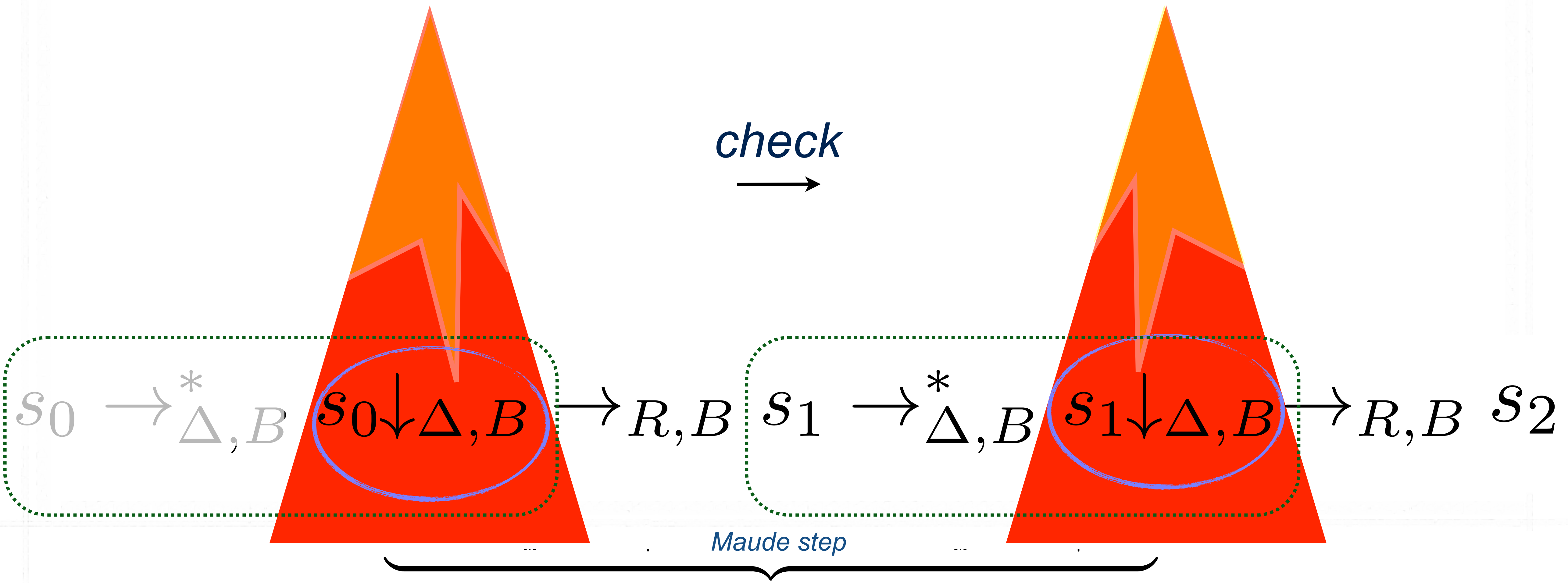
Slicing criterion

< ● | ● | blocked >

Assertion-based slicing

Check incrementally processes the *Maude steps* of a trace, while *checking*

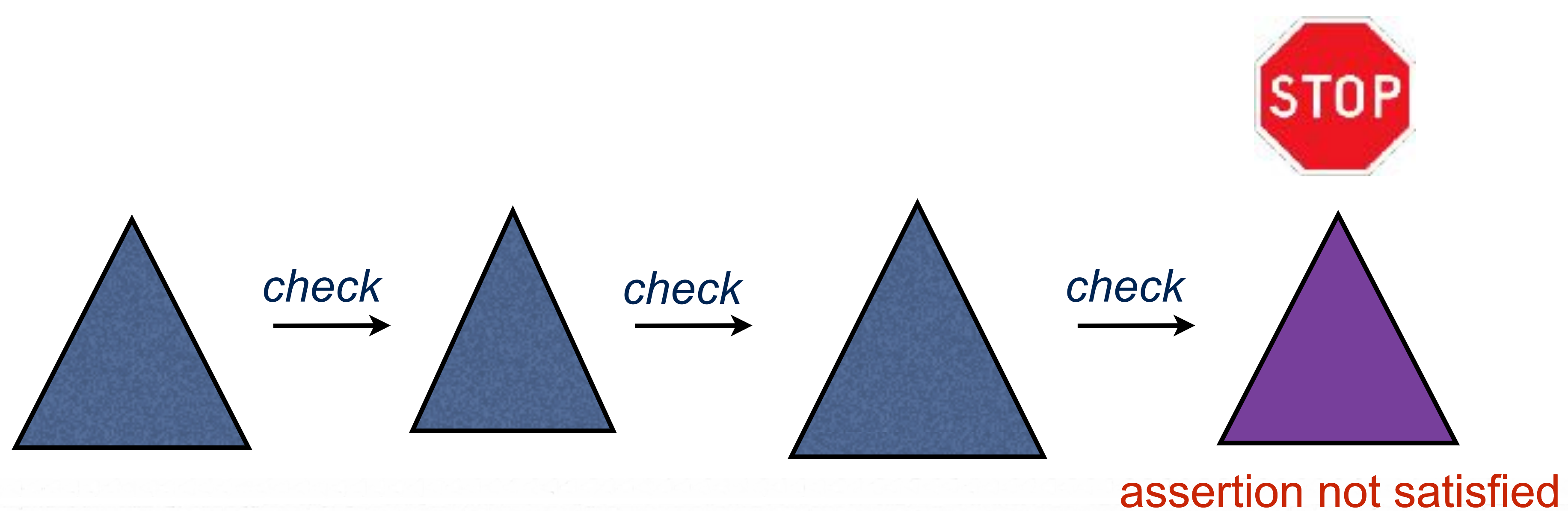
- the functional assertions, at each state normalization $s \rightarrow^{*}_{\Delta,B} s{\downarrow}_{\Delta,B}$
- the sytem assertions, at each (normalized) state $s{\downarrow}_{\Delta,B}$



Assertion-based slicing

- The checker proceeds *from front to back*
- In the event that an assertion is falsified, it automatically *halts*

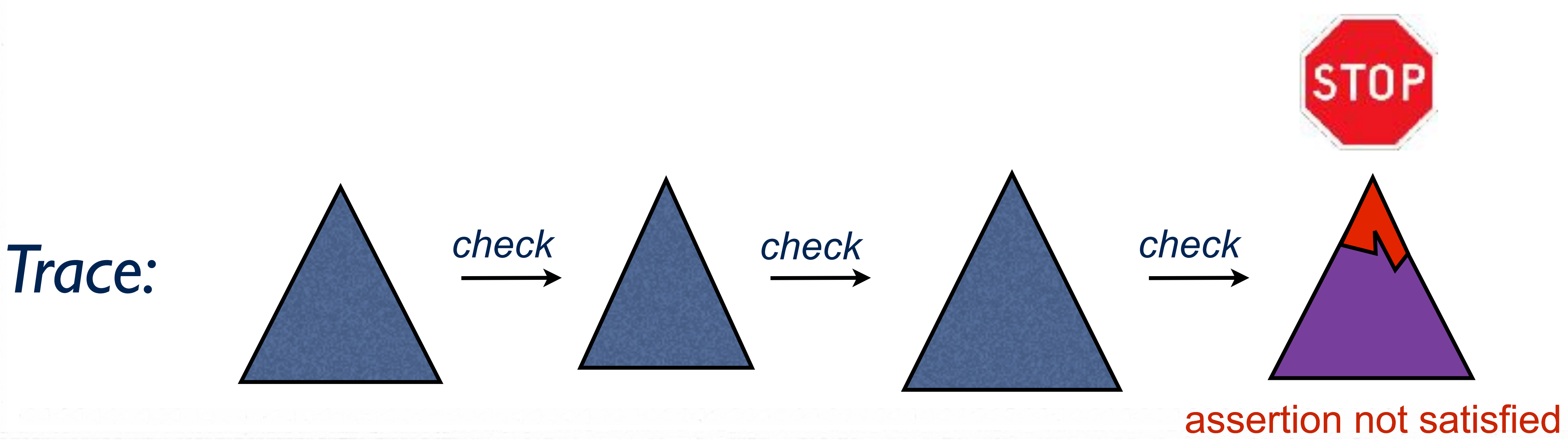
Trace:



Assertion-based slicing

- The checker proceeds *from front to back*
- In the event that an assertion is falsified, it automatically *halts*

1. The error symptoms are *distilled* and *translated* into suitable slicing criteria

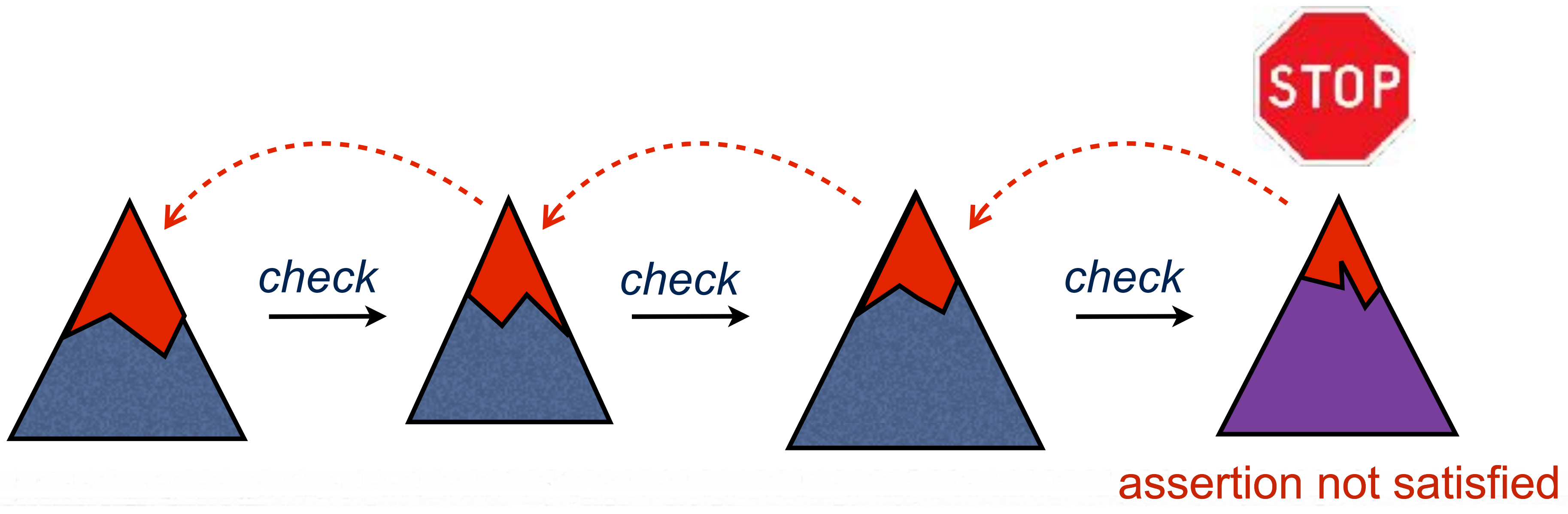


Assertion-based slicing

- The checker proceeds *from front to back*
- In the event that an assertion is falsified, it automatically *halts*

1. The error symptoms are *distilled* and *translated* into suitable slicing criteria
2. The slicer is run to compute the smallest fragment of the trace that highlights the wrong behavior

Trace:



The ABETS system

- Written in (a custom version of) Maude 2.7, with a web GUI

  Several maude operations have been directly coded into native C functions.

- Available as a web application at

  http://safe-tools.dsic.upv.es/abets/

- Synchronous (on-line) and asynchronous (off-line) analysis

- Extended to (full) Maude computations

Conclusions

- Dynamic techniques and tools that helps developers understand and debug (Full) Maude programs
    - program animation
    - backward trace slicing
    - assertion-driven backward trace slicing

Future work

- Work on static methodologies to enforce safety constraints on Maude programs

Assertions + Program specialization

to infer safe Maude programs

References

- M. Alpuente, D. Ballis, F. Frechina, D. Romero:
  Using conditional trace slicing for improving Maude programs.
  SCP 80: 385-415 (2014)

- M. Alpuente, D. Ballis, F. Frechina, J. Sapiña:
  Exploring conditional rewriting logic computations.
  JSC 69: 3-39 (2015)

- M. Alpuente, D. Ballis, F. Frechina, J. Sapiña:
  Debugging Maude programs via runtime assertion checking and trace slicing.
  JLAMP 85(5): 707-736 (2016)

- M. Alpuente, F. Frechina, J. Sapiña, D. Ballis:
  Assertion-based analysis via slicing with ABETS.
  TPLP 16(5-6): 515-532 (2016)