debugging in domain-specific modelling

raphaël mannadiar and hans vangheluwe

presented by hans vangheluwe

SLE 2010

outline

outline

why do domain-specific modelling (dsm)?

problem and solution domains are often far apart

mapping problems to solutions manually is difficult, slow and error-prone

but the process can be automated!

why do domain-specific modelling (dsm)?

problem and solution domains are often far apart

mapping problems to solutions manually is difficult, slow and error-prone

but the process can be automated!

dsm allows domain experts to play active roles in the development process, even if they aren't solution domain experts

what's under the hood?

artifacts are generated from domain-specific models (ds*m*s)

artifacts may be configuration files, programs, performance models, etc.

rules of the trade dictate that artifact generation should be done via model transformations

how can i play?

the typical steps of a dsm project are...

1. modelling a domain-specific language (dsl)
2. specifying its semantics as model transformations
3. creating instance dsms
4. synthesizing artifacts from dsms

what can go wrong?

bugs may creep in at any stage of a dsm project

means to debug model transformations, ds*ms* and synthesized artifacts are necessary!

has anyone tried?

common approach to debugging ds*m*s

debug **coded artifacts** to debug ds*m*s

$\equiv$

debug **bytecode** to debug a coded program.

has anyone tried?

common approach to debugging ds*m*s

debug **coded artifacts** to debug ds*m*s

$\equiv$

debug **bytecode** to debug a coded program.

a better approach by wu et al.

build **mapping** between **domain-specific code statements** and **artifact code statements** during generation

combine mapping with Eclipse plugin that uses Eclipse debugger

$\rightarrow$

debugging is performed **directly on ds*m*s**

what's missing?

nothing is available for debugging visual ds*ms* (not textual)

nothing is available for debugging modelled artifacts (not coded)

nothing is available for debugging model transformations

what's missing?

nothing is available for debugging visual ds*ms* (not textual)

nothing is available for debugging modelled artifacts (not coded)

nothing is available for debugging model transformations

we propose a mapping between debugging concepts (e.g., breakpoints, assertions) in the software and dsm realms meant as a guide for developing complete debuggers for dsm

outline

debugging code 101

the bases of software debugging are observing system state and hand-simulation

facilities to do this are bundled within modern programming languages and integrated development environments

language facilities

print statements

- output variable contents
- monitor program flow

language facilities

print statements

- output variable contents
- monitor program flow

assertions

- verify runtime conditions
- compiler enabled/disabled

language facilities

print statements

- output variable contents
- monitor program flow

assertions

- verify runtime conditions
- compiler enabled/disabled

exceptions

- indicate and describe problematic system state
- propagated or handled

ide facilities

execution control

- play, step, pause, stop
- step over, step into, step out
- release vs. debug modes

ide facilities

execution control

- play, step, pause, stop
- step over, step into, step out
- release vs. debug modes

runtime variable i/o

- read/write global/local variables

ide facilities

execution control

- play, step, pause, stop
- step over, step into, step out
- release vs. debug modes

runtime variable i/o

- read/write global/local variables

breakpoints

- pause on marked statement

ide facilities

execution control

- play, step, pause, stop
- step over, step into, step out
- release vs. debug modes

runtime variable i/o

- read/write global/local variables

breakpoints

- pause on marked statement

stack traces

- navigable call stack to current statement

debugging in dsm 101

the development process in dsm has two important facets: developing **models** and developing **model transformations**

this introduces two important differences with the programming world:

1. **artifacts** to debug are **not restricted to code**
2. designing and debugging **"compilers/interpreters"** is now common

debugging in dsm 101...

in the paper, we explore how all of the above language and debugging facilities translate into the debugging stages of both facets of dsm development

in this talk, we focus only on **print statements**, **exceptions** and **execution control**

outline

dsm and model transformations

model transformations can elegantly define a ds*m*'s **semantics**

we focus only on **rule-based** model transformations

a rule-based model transformation describes a **flow of rules*** (which may require debugging)

print statements and model transformations

naive approach: normal rules with output calls in action code

- requires identical source and destination patterns
- requires loop prevention means

→ lots of accidental complexity

print statements and model transformations

naive approach: normal rules with output calls in action code

- requires identical source and destination patterns
- requires loop prevention means

→ lots of accidental complexity

domain-specific approach: extend model transformation languages with *print rules*

- parameterized with one pattern, condition code and *printing code*
- easily transformed to above naive rules

→ no accidental complexity

print statements and model transformations...

it may seem odd to support a language construct whose usefulness is mostly restricted to debugging purposes, but...

we should remember that print statements (whose usefulness is mostly restricted to debugging purposes) are supported in every modern GPL

stepping and model transformations

stepping over should run one (possibly composite) rule

stepping into should run one sub-rule (if any), or one primitive operation (in t-core based systems*)

stepping out should run in continuous mode until scope change

some modern tools (e.g., atom^3) support basic rule-by-rule execution

stepping and model transformations

stepping over should run one (possibly composite) rule

stepping into should run one sub-rule (if any), or one primitive operation (in t-core based systems*)

stepping out should run in continuous mode until scope change

some modern tools (e.g., atom[3]) support basic rule-by-rule execution

stepping is heavily dependent on model transformation language and engine features

pausing and model transformations

naive approach

immediate interruption

pausing and model transformations

naive approach

immediate interruption

transactional approach

commit/roll-back current rule or t-core operation before pausing

pausing and model transformations

naive approach
immediate interruption

transactional approach
commit/roll-back current rule or t-core operation before pausing

pausing should only occur when the system state is consistent and observable

pausing is also heavily dependent on model transformation language and engine features

outline

dsm, models and artifacts

running models with denotational semantics implies executing synthesized artifacts (as opposed to model transformations) and observing ds*m* versus artifact evolution

in industry, dsls (and their semantics) might be defined by different actors than the end-users

thus, we distinguish between two types of users

dsm, models and artifacts

running models with denotational semantics implies executing synthesized artifacts (as opposed to model transformations) and observing ds*m* versus artifact evolution

in industry, dsls (and their semantics) might be defined by different actors than the end-users

thus, we distinguish between two types of users

designers

are fully aware of the model transformations that describe model semantics and generate artifacts

dsm, models and artifacts

running models with denotational semantics implies executing synthesized artifacts (as opposed to model transformations) and observing ds*m* versus artifact evolution

in industry, dsls (and their semantics) might be defined by different actors than the end-users

thus, we distinguish between two types of users

designers

are fully aware of the model transformations that describe model semantics and generate artifacts

modellers

have implicit understanding of model semantics but little or no knowledge about how they are specified

exceptions, models and artifacts

ds*m*s might be animated but what is truly being executed are synthesized artifacts

$\rightarrow$ exceptions originate from artifacts

exceptions, models and artifacts

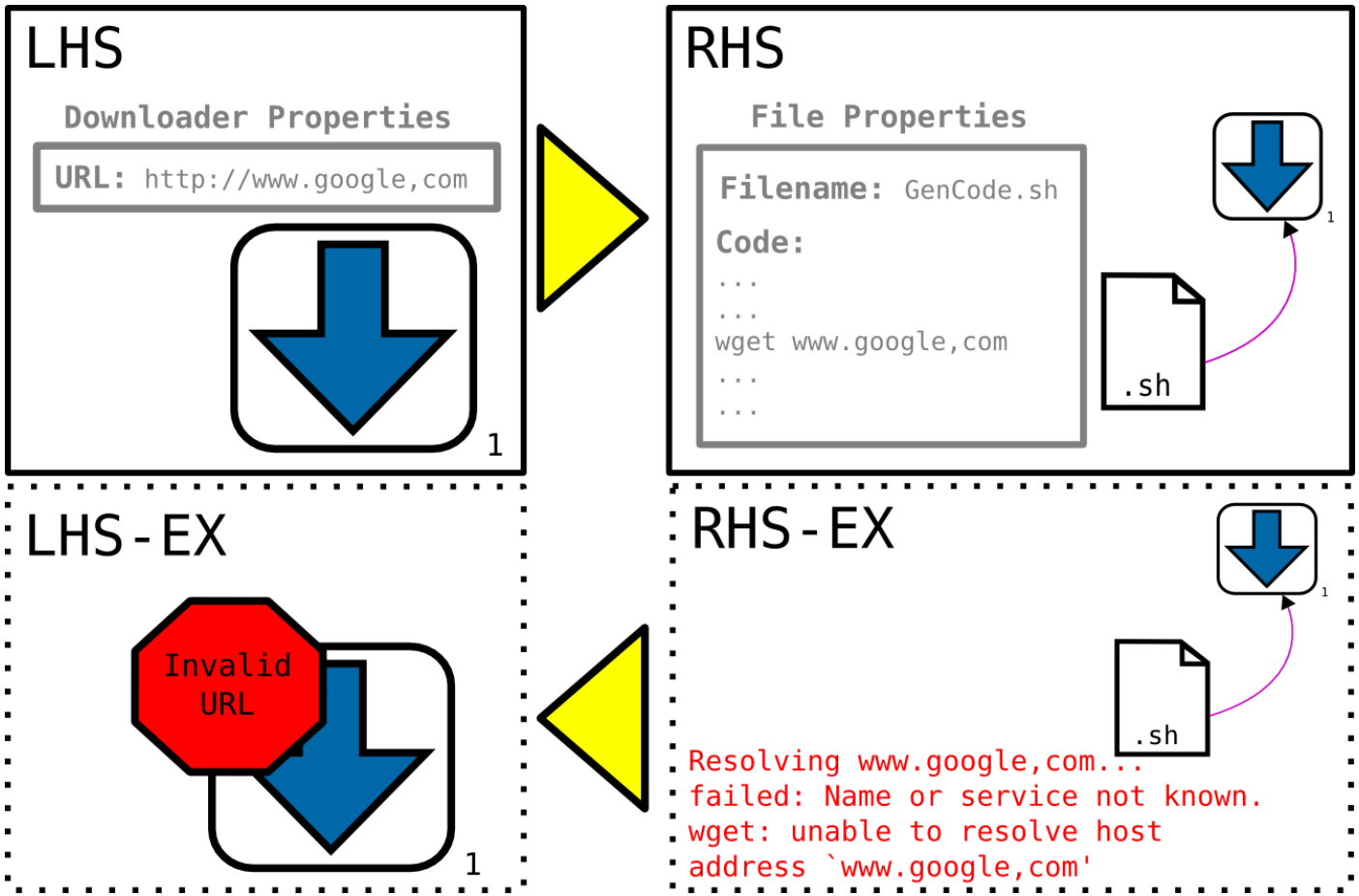
ds*m*s might be animated but what is truly being executed are synthesized artifacts

$\rightarrow$ exceptions originate from artifacts

which exceptions to catch and propagate are design decisions of the dsl architect

- "silent" handlers for irrelevant exceptions should be generated alongside synthesized artifacts
- relevant exceptions should be translated into domain-specific terms and propagated to the modeller or designer

exceptions, models and artifacts...



stepping, models and artifacts

what is a step in an arbitrary ds*m*?

stepping, models and artifacts

what is a step in an arbitrary ds*m*?

we define a step as any modification to any parameter of any entity in a ds*m*

stepping, models and artifacts

what is a step in an arbitrary ds*m*?

we define a step as **any modification to any parameter of any entity in a ds*m***

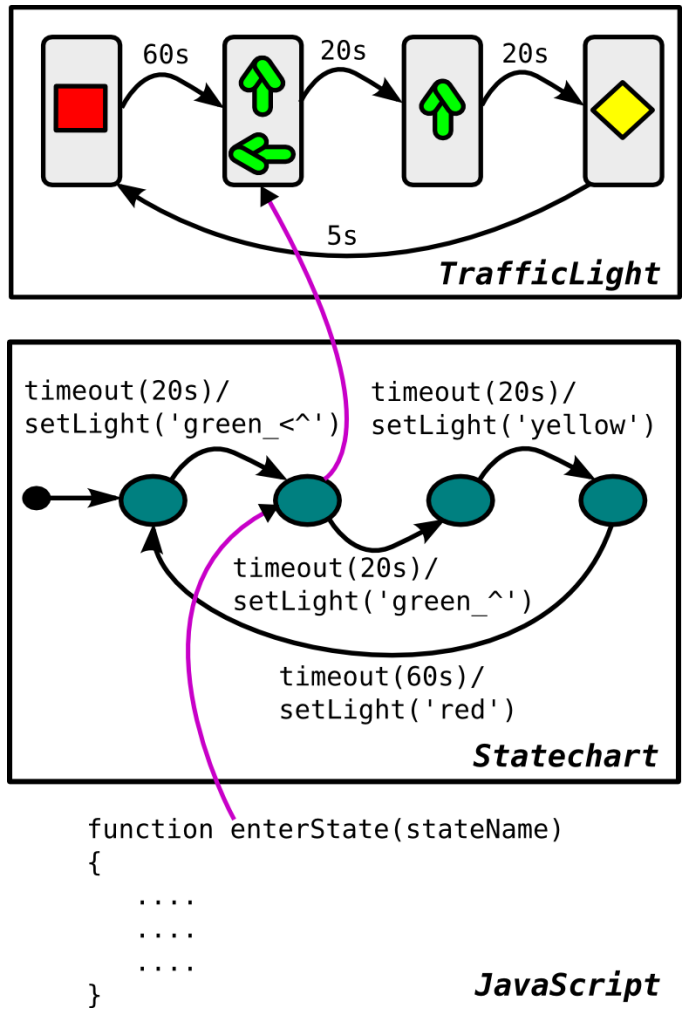
stepping can be considered from **two orthogonal perspectives**: the modeller's and the designer's

for modellers, the three step commands intuitively translate to dsls with hierarchy and composition

stepping, models and artifacts...

generating artifacts from ds*m*s creates an **implicit hierarchy** between them

a designer may prefer for the step into operation to **take a step** at the level of **corresponding lower level entities***



pausing, models and artifacts...

a sensible approach is to pause the execution before running...

- the next step at the ds*m* level for the modeller
- the next step at the artifact level for the designer

a key enabler for pausing and stepping is (ideally automatic) instrumentation of artifacts to enable running only parts of them at a time

outline

sales pitch

our work is meant as a guide for developing complete debuggers for dsm

sales pitch

our work is meant as a guide for developing complete debuggers for dsm

in the paper, you'll find

- a detailed mapping for all of the listed debugging concepts
- a clear distinction between the debugging of model transformations and ds*m* debugging, and between debugging scenarios for *designers* and *modellers*
- a discussion on how to generate more readily debuggable artifacts

questions?

thank you!

ds*m* to artifact correspondence links