Declarative program development in Prolog with GUPU

Ulrich Neumerkel, Stefan Kral
Technische Universität Wien

• Programming environment for beginners
• New program development process
  specification & implementation phase
• All phases are supported by diagnostic facilities
• Emphasizing notion of relation
1,
GUPU

Gesprächs
GUPU

Gesprächsunterstützende 1, 2, 3.
GUPU

Gesprächsunterstützende Programmierübungs

1, 2, 3, 4.
GUPU

Gesprächsunterstützende Programmierübungsurngebung

1, 2, 3, 4, 5,
GUPU — explication

Gesprächsunterstützende Programmierübungsübungsumgebung

environnement
GUPU — explication

Gesprächsunterstützende Programmierübungsurngebung
programmation ⇐ cours ⇐ environment
GUPU — explication

Gesprächsunterstützende Programmierübungsprogrammierungssupportant ← programmation ← de cours ← de environment
GUPU — explication

Gesprächsunterstützende Programmierübungsumgebung conversations des supportant ← programmation de cours de environment
GUPU — explication — explanation

Gesprächsunterstützende Programmierübungsgrundgebung

conversations des supportant de programmation de cours de environnement

Conversation

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
GUPU — explication — explanation

Gesprächsunterstützende Programmierübungsprogrammierung
conversations des supportant de programmation de cours de environnement
Conversation → supporting
GUPU — explication — explanation

Gesprächsunterstützende Programmierübungs
conversations ← supportant ← programmation ← cours ← enironnement
Conversation → supporting → programming
GUPU — explication — explanation

Gesprächsunterstützende programmiereübungsumgebung
conversations des supportant ← programmation de cours de environnement
Conversation → supporting → programming → course
GUPU — explication — explanation

Gesprächsunterstützende
conversations \( \leftarrow \) supportant \( \leftarrow \) programmation \( \leftarrow \) cours \( \leftarrow \) environnment
Conversation \( \rightarrow \) supporting \( \rightarrow \) programming \( \rightarrow \) course \( \rightarrow \) environment
GUPU — explication — explanation

Gesprächsunterstützende Programmierübungs umgebung
conversations des supportant de programmation de cours de environnment
Conversation → supporting → programming → course → environment

• Used since 1992
• Under continual development since 1991
• Original motivation: realize courses with a large number of students
• Eases assessment (marking) — instantaneous, automated pre-marking
• General attitude: Mark now, don’t delay it unto the end
• 9 weeks/about 80 (small) exercises
• Flexible low cost system for deadlines
• Simple to use — very simple interaction mode
• Consistent view of program
• Useless notions absent (files, shells, overlapping windows etc.)
• Side effect free. Pure, monotone subset of Prolog including constraints
• currently trilingual (German, French, English)
Challenge: Understanding relations

- easy to confuse with procedures
- Algorithm = Logic + Control.

Conclusion:
Challenge: Understanding relations

• easy to confuse with procedures

• Algorithm = Logic + Control. Often: Algorithm ≈ Control

• Conclusion:
Challenge: Understanding relations

- easy to confuse with procedures

- Algorithm = Logic + Control. Often: Algorithm \( \approx \) Control \( \Rightarrow \) Logic \( \approx 0 \)

- Conclusion:
Challenge: Understanding relations

- easy to confuse with procedures
- Algorithm = Logic + Control. Often: Algorithm ≈ Control ⇒ Logic ≈ 0
- input/output:

- Conclusion:
Challenge: Understanding relations

- easy to confuse with procedures
- Algorithm = Logic + Control. Often: Algorithm \approx Control \Rightarrow Logic \approx 0
- input/output:
  What is the input/output of a predicate?
  Answer:

- Conclusion:
Challenge: Understanding relations

- easy to confuse with procedures
- Algorithm = Logic + Control. Often: Algorithm \(\approx\) Control \(\Rightarrow\) Logic \(\approx\) 0
- input/output:
  What is the input/output of a predicate?
  Answer: There is no input/output — not helpful.

- Conclusion:
Challenge: Understanding relations

• easy to confuse with procedures

• Algorithm = Logic + Control. Often: Algorithm \approx Control \Rightarrow Logic \approx 0

• input/output:
  What is the input/output of a predicate?
  Answer: There is no input/output — not helpful.

• Naming: avoid imperative names — helps somewhat, but soon forgotten.

• Conclusion:
Challenge: Understanding relations

- easy to confuse with procedures
- Algorithm = Logic + Control. Often: Algorithm \( \approx \) Control \( \Rightarrow \) Logic \( \approx \) 0
- input/output:
  What is the input/output of a predicate?
  Answer: There is no input/output — not helpful.
- Naming: avoid imperative names — helps somewhat, but soon forgotten.
- Programming techniques — currently no experience.

Conclusion:
Challenge: Understanding relations

- easy to confuse with procedures
- Algorithm = Logic + Control. Often: Algorithm ≈ Control ⇒ Logic ≈ 0
- input/output:
  What is the input/output of a predicate?
  Answer: There is no input/output — not helpful.
- Naming: avoid imperative names — helps somewhat, but soon forgotten.
- Programming techniques — currently no experience.
- Read predicates as English sentences — works only for very tiny programs.
- Conclusion:
Challenge: Understanding relations

- easy to confuse with procedures
- Algorithm = Logic + Control. Often: Algorithm \(\approx\) Control \(\Rightarrow\) Logic \(\approx 0\)
- input/output:
  What is the input/output of a predicate?
  Answer: There is no input/output — not helpful.
- Naming: avoid imperative names — helps somewhat, but soon forgotten.
- Programming techniques — currently no experience.
- Read predicates as English sentences — works only for very tiny programs.
- Selective reading of predicates — works also for larger programs.
- Conclusion:
Challenge: Understanding relations

- easy to confuse with procedures
- Algorithm = Logic + Control. Often: Algorithm \( \approx \) Control \( \Rightarrow \) Logic \( \approx \) 0
- input/output:
  What is the input/output of a predicate?
  Answer: There is no input/output — not helpful.
- Naming: avoid imperative names — helps somewhat, but soon forgotten.
- Programming techniques — currently no experience.
- Read predicates as English sentences — works only for very tiny programs.
- Selective reading of predicates — works also for larger programs.
- Conclusion: If predicates are written, it is already too late.
Challenge: Understanding relations

- easy to confuse with procedures
- Algorithm = Logic + Control. Often: Algorithm $\approx$ Control $\Rightarrow$ Logic $\approx$ 0
- input/output:
  - What is the input/output of a predicate?
  - Answer: There is no input/output — not helpful.
- Naming: avoid imperative names — helps somewhat, but soon forgotten.
- Programming techniques — currently no experience.
- Read predicates as English sentences — works only for very tiny programs.
- Selective reading of predicates — works also for larger programs.
- Conclusion: If predicates are written, it is already too late.

How to focus on the declarative properties?

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Extreme Programming

Lightweight, agile method developed by Kent Beck for Smalltalk. Practice to **Code Unit Test First**  *Test program into existence!*

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often.
Extreme Programming

Lightweight, agile method developed by Kent Beck for Smalltalk.
Practice to **Code Unit Test First**  *Test program into existence!*

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often.

In LP tests are

- *much* easier to write  ... than in traditional languages
- *much more expressive* ... “”

because of ...
Extreme Programming

Lightweight, agile method developed by Kent Beck for Smalltalk. Practice to **Code Unit Test First**  *Test program into existence!*

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often.

In LP tests are

- *much* easier to write ...
- *much* more expressive ...

because of **logical variables**.

- less specific: ← alldifferent(Xs).
- higher coverage: ∉ alldifferent([X,X|_;]).
Focus on declarative properties in GUPU

- Tests: assertions
  - Positive assertions: \( \leftarrow \text{Goal} \) should succeed
  - Negative assertions: \( \not\leftarrow \text{Goal} \) should fail
- Close integration: Tests are written \textit{into the program text}
- All predicates must have assertions
- Errors are signaled immediately \textit{within the program text}, explanations based on \textit{slicing} are offered
- Adding further assertions very easy
  - Duplicate and modify existing assertion
  - Offered by diagnostic facilities
- Tests are run very often: Upon every saving, all assertions are tested
Methodology for writing assertion tests

1. Start with the least specific test.
   \[ \text{\texttt{alldifferent(Xs).}} \text{ There is at least a single solution, \textit{Xs is anything}} \]

2. Estimate cardinality of \textit{minimal} possible set of answer substitutions.
   If infinite, goal \textit{must not} terminate.
   \[ \text{\texttt{\neg alldifferent(Xs), false.}} \]

3. Go further to more specific tests.
   \[ \text{\texttt{Xs = [.,], alldifferent(Xs).}} \]

4. For every positive assertion, find a similar negative assertion.
   \[ \text{\texttt{\neg Xs = [1,1], alldifferent(Xs).}} \]

5. Generalize negative assertions as much as possible.
   \[ \text{\texttt{\neg Xs = [X,X], alldifferent(Xs).}} \]

6. Specialize positive assertions as much as possible.
   \[ \text{\texttt{Xs = [1,2], alldifferent(Xs).}} \]

But, one problem remains...
Testing prior to coding

Biggest obstacles to testing prior to coding:

- Cumbersome to write tests containing lots of data
- Incorrect tests slows development
- No motivation to write tests since they might be wrong
- Adjusting tests to the program

Conclusion:
Testing prior to coding

Biggest obstacles to testing prior to coding:

- Cumbersome to write tests containing lots of data
- Incorrect tests slows development
- No motivation to write tests since they might be wrong
- Adjusting tests to the program

Conclusion:

- **Attention span too large** for beginners

Solution:

- Put learner into the position of testing predicates prior to writing them
Reference implementation — testing the tests

Assertions are tested against reference implementation. Reference implementation is considered correct for:

- unconditional success (no pending constraints)
- finite failure

Reference implementation is ignored for:

- (implementation related — reference implementation not perfect)
  - exceptions
  - computation takes too long/loops
  - conditional success with constraints that cannot be resolved
- (specification related — relation is under-specified)
  Signaled as exceptions or constraints. E.g.: child_of/2

⇒ All procedural issues are ignored.
Marking system already counts correct assertions.
Diagnosis of incorrect negative assertions

• reason: there is a solution

• show solution in the form of a positive assertion

• try to make assertion as specific as possible
  – show binding (answer substitution)
  – try to ground remaining variables with constants any1, ...
    \( \neg Xs = [-,-,\ldots], \text{alldifferent}(Xs). \)
    @@ % != Should be positive!
    @@ % Even this specialized assertion should be true
    @@ \( \leftarrow Xs = [\text{any0,any1,any2}], \text{alldifferent}(Xs). \)
  – try to ground fd-variables with some values
Diagnosis of incorrect positive assertions

- reason: there is no solution
- show a generalized goal in form of a negative assertion
- try to generalize assertion to better localize the error

← alldifferent([a,b,c,d,c,f]).
@@ % != Should be negative!
@@ % @ Generalized negative assertion
@@ \(\neg\) alldifferent([\_,\_,c,\_,c,\_]).
@@ % @ Further generalization
@@ \(\neg\) alldifferent([\_,\_,V0,\_,V0,\_]).
@@ % @ Generalization by goal replacement
@@ \(\neg\) alldifferent([V0,\_,V0|\_]).
← [a,b,c,d] = [a,b,e,d].
@@ % != Fails as it should!
@@ % @ Generalized negative assertion
@@ \(\neg\) [\_,\_,c|\_]=\[\_,\_,e|\_].
@@ % @ Generalization using dif/2
@@ \(\neg\) dif(V0,V1),[\_,\_,V0|\_]=[\_,\_,V1|\_].
Some revealing examples

code_inconnu/2:

• Nothing is said about the relation except that you will only get information about it via assertions

• Relation defined differently for everyone

Effects of testing with reference implementation

+ test coverage significantly better

+ more than twice as many assertions are written

+ almost no incorrect programs (i.e. automatic marking almost perfect)

+ students consider (and question) the example statements more closely

+ almost no student questions concerning example statements (most frequent question previously: What is the output?)

+ (the very few) questions focus rather on the specification itself

+ more fun due to fast response
After coding: reading of programs

• traditional readings: declarative and procedural

• selective readings: use transformations to obtain slices (fragments)

**generalization:** delete goals

father(Father) ←
* male(Father),
  child_of(_C, Father).

**specialization:** add goals (false/0: failure slice).

married_to(Husband, Spouse) ← false,
  husband_spouse(Husband, Spouse).
married_to(Spouse, Husband) ←
  husband_spouse(Husband, Spouse).

+ eases reading of larger programs
+ remains close to source code, simple presentation by hiding parts
+ no new formalism like proof trees, traces
+ works also with incomplete constraints
Slicing explanations

**insufficiency (unexpected success):** maximal failing generalization explains *data inconsistency* and *modeling errors*

**incorrectness (unexpected success):** maximal specialization (with false/0) that succeeds

**non-termination:** maximal non-terminating specialization

Common properties:

+ error in fragment implies error in original program
+ visible fragment has to be changed
+ no user-interaction ($\Rightarrow$ no debugging errors possible!)

? *slicing* or *program modification* ?
Viewers

- side effect free visualization of answer substitutions
- general form: $\leftarrow$ Viewer $\llll$ Goal.
- $\llll$ can only be used within assertions, not allowed in rules
- most viewers are implemented side effect free within GUPU
- very few elementary viewers text, postscript
Problems searching for explanations of unexpected failure

- non-termination because of generalized fragments
  → analyse termination (cTI)

- complexity: sub-problem already NP-hard, no approximation possible
  \(\text{(Monotone Minimum Satisfying Assignment), Umans 1999}\)
  → search local minima, one by one (one test per line)

- labeling for generalized fragments often very expensive
  → adopt labeling strategy

Similar sub-problem: Explanations in PPC (Narendra Jussien)

- generalization of (dynamic) constraint system
- much more constraints (at runtime) than (static) program points
  - more costly
  - less readable — but contains more information
- uses a search interlaced with labeling (very interesting!)