DOI: 10.15514/ISPRAS-2024-36(5)-4

CC BY

Декларативный синтез графических интерфейсов пользователя с помощью реляционного решателя ограничений

Д.С. Косарев, ORCID: 0000-0002-6773-5322 <d.kosarev@spbu.ru>
П.А. Лозов, ORCID: 0000-0003-3563-2828 <lozov.peter@gmail.com>
Д.Ю. Булычев, ORCID: 0000-0001-8363-7143 <dboulytchev@math.spbu.ru>

Санкт-Петербургский государственный университет,
Россия, 198504, Санкт-Петербург, Университетский пр., д. 28.

Аннотация. Авторы представляют систему, которая по набору правил-ограничений на дизайн и по структурному описанию пользовательского интерфейса (GUI), порождает набор конкретных интерфейсов, каждый из которых по построению соблюдает эти ограничения. Задача, ставящаяся перед системой, описывается как проблема удовлетворения ограничений, после чего на основе реляционного подхода “решатель-из-верификатора” конструируется корректный и полный решатель. Также описывается набор улучшений, делающих предложенный решатель более эффективным.

Ключевые слова: проектирование интерфейсов; синтез программ; программирование в ограничениях; реляционное программирование; встраиваемый язык miniKanren.

Для цитирования: Косарев Д.С., Лозов П.А., Булычев Д.Ю. Декларативный синтез графических интерфейсов пользователя с помощью реляционного решателя ограничений. Труды ИСП РАН, том 36, вып. 5, 2024 г., стр. 47–66. DOI: 10.15514/ISPRAS–2024–36(5)–4.

Declarative GUI Layout Synthesis with Relational Constraint Solvers

D.S. Kosarev, ORCID: 0000-0002-6773-5322 <d.kosarev@spbu.ru>
P.A. Lozov, ORCID: 0000-0003-3563-2828 <lozov.peter@gmail.com>
D.Yu. Boulytchev, ORCID: 0000-0001-8363-7143 <dboulytchev@math.spbu.ru>

St. Petersburg State University,
Universitetski pr., 28, St. Petersburg, 198504, Russia.

Abstract. Authors describe a system which, given a set of designer-specified layout constraints (guidelines) and a description of graphic user interface (GUI) logical structure generates a set of particular layouts. Each of these layouts comply with given guidelines by construction. Authors also give a formal treatment of the task as a constraint satisfiability problem and describe the construction of a sound and complete solver based on the utilization of relational verifier-to-solver approach. They also describe a number of refinements which make the solver more efficient and applicable.

Keywords: interface design; program synthesis; constraint programming; relational programming; embedded domain-specific language miniKanren.

For citation: Kosarev D.S., Lozov P.A., Boulytchev D.Yu. Declarative GUI Layout Synthesis with Relational Constraint Solvers. Trudy ISP RAN/Proc. ISP RAS, vol. 36, issue 5, 2024. pp. 47-66 (in Russian). DOI: 10.15514/ISPRAS-2024-36(5)-4.

1. Введение

Графические интерфейсы пользователя (GUI) – один из самых распространенных способов взаимодействия с программами. Как на персональном компьютере, так и на ноутбуке, мобильном телефоне, банкомате, игровой консоли и т.д. пользователь встретится с некоторым набором графических элементов управления и интуитивно понятным смыслом. За такой низкий порог вхождения мы платим некоторую цену: графические интерфейсы управления не так просто разрабатывать [1, 2].

Обычные пользователи как правило не осознают, что делает графический интерфейс приятно выглядящим и удобным в использовании. Чтобы получить качественный интерфейс необходимо учесть множество эстетических, эргономических и психологических аспектов, а это требует экспертизы, которой программисты обладают не всегда. Из-за этого появляются должности не только по разработке интерфейсов (UI), но и по дизайну взаимодействия с пользователем (UX) [3], где специалисты разрабатывают правила (англ. guidelines) разработки хороших интерфейсов. Данные правила часто неполны, двусмысленны и неформальны, а следование им требует интуиции, которой разработчики могут и не обладать. Взаимодействие между разработчиками и дизайнерами часто осложняется разницей в экспертизе, подходах и складах мышления [4]. Более того, упомянутые правила не всегда естественно переносятся между разными устройствами и даже разрешениями экрана. Поэтому, чтобы поддерживать все желаемые платформы, разработчикам придется следовать нескольким наборам правил проектирования интерфейсов одновременно.

В данной работе представлен подход к синтезу расположения (англ. layout) элементов управления GUI с учетом набора правил проектирования интерфейсов. Наша модель GUI отделяет логическую структуру элементов управления (часть предельно понятная программистам), от конкретного расположения элементов, отступов и выравниваний (понятных дизайнеру). Мы разработали автоматический подход, который по логической структуре (предоставляемой программистом) и набору правил (предоставленных дизайнером) синтезирует раскладку элементов управления, удовлетворяющую этим правилам. В общем случае правила неоднозначны, поэтому несколько конкретных раскладок может им удовлетворять. Мы рассматриваем синтез интерфейсов как задачу удовлетворения

ограничений (англ. constraint satisfaction), и используем реляционное программирование [5] и подход верификатор-из-решателя [6] (англ. verifier-to-solver), чтобы получить решатель, располагающий элементы управления с помощью примитивов. В примитивной форме такой решатель неэффективен, и мы применили набор оптимизаций (в том числе специализацию), чтобы его ускорить. В результате получился решатель, написанный на смеси из реляционного и функционального кода. Затем набор примитивов расположения превращается в линейные ограничения на координаты и решается с помощью SMT решателя Z3 [7], что дает нам абсолютные координаты всех элементов управления. Этот подход позволяет получить *корректный* и *полный* решатель: по указанной логической структуре он синтезирует все расположения элементов, удовлетворяющие правилам дизайна.

При этом решатель достаточно эффективен, чтобы запускаться на обычном ноутбуке.

2. Модель GUI

В данном разделе мы изложим модель пользовательского интерфейса, которая учитывает все важные особенности предметной области. Для реального использования модель должна быть немного расширена, но это не предоставит сложности, так как расширения не вносят серьёзных изменений в алгоритм синтеза.

Мы можем выделить три важных понятия, связанных с интерфейсами пользователя: *структура*, *расположение* (англ. layout) и набор правил проектирования интерфейсов (англ. guideline), который для краткости далее будем называть *предписанием*. *Структура* описывает набор элементов управления и связи между ними, *расположение* уточняет их визуальные позиции друг к другу. Такое разделение согласуется с распространенной практикой отделения бизнес-логики от визуального представления [8]. Наконец, *предписание* отображает определенные структуры в конкретные примитивы расположения.

Продемонстрируем эти понятия на примере. Рассмотрим интерфейс с рис. 1 и опишем структуру следующим образом.

- Всего три элемента управления GUI: галочка (англ. check box), метка и поле со списком (англ. combo box).
- Присутствует зависимость между меткой и полем, так как первое *описывает* второе; следовательно, их можно объединить в одну сущность.
- Существует зависимость между галочкой и только что введенной композитной сущностью, так как, по-видимому, дизайнер интерфейса полагал, что важность элементов управления убывает сверху вниз.

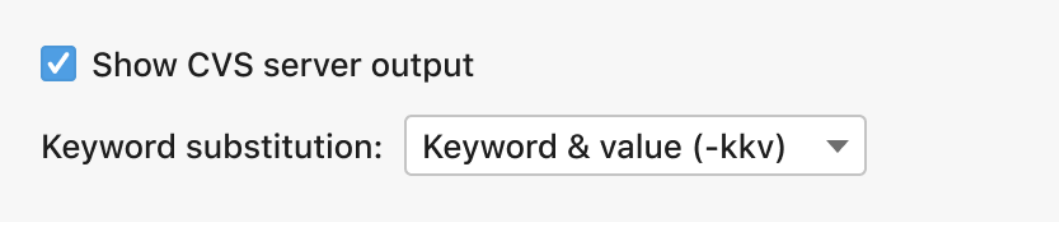


Рис. 1. Пример интерфейса GUI.
Pic. 1. Sample GUI form.

Мы схематично изобразили структуру на рис. 2. Теперь мы можем рассматривать её как набор именованных *отношений* между элементами управления, или даже над данными в них (строками, числами и т.п.). Наш подход разделяет все отношения на две категории: *абстрактные* и *конкретные*. Семантика абстрактных определяется предписанием, конкретных – нашей системой. Конкретные описывают типы элементов управления, различные их свойства (содержимое текстовых полей, размеры, и т.п.), а также позволяют

объединять элементы управления в упорядоченные или неупорядоченные группы, или в *виртуальные* – элементы управления, объединяющие несколько в один целый.

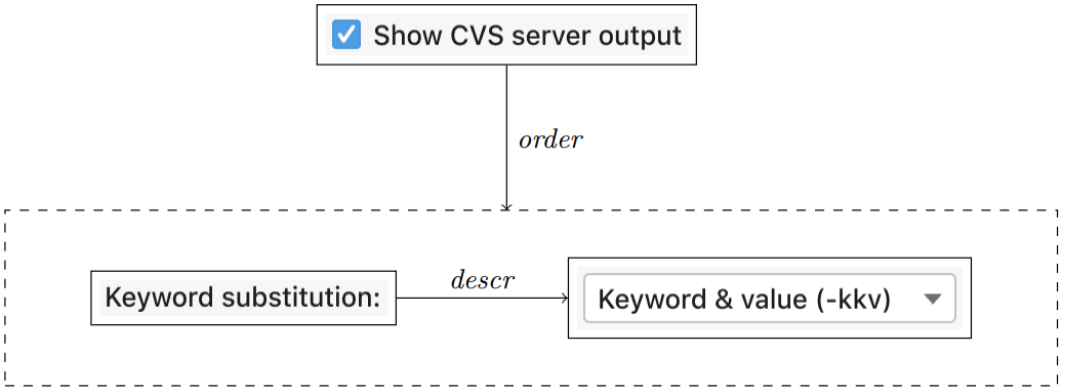


Рис. 2. Пример структуры GUI.
Pic. 2. Sample GUI form structure.

Множество *абстрактных* отношений включает: бинарное упорядочивание (один-за-другим), описание (один описывает другой, например, метка поле ввода или галочку), подчиненность (галочка делает активным/неактивным поле ввода). В реализации эти отношения – это просто удобные имена, используемые в структуре и предписании. Их семантика полностью определяется предписанием, а система никаким особым образом не обрабатывает эти отношения. Пользователи могут добавлять новые абстрактные отношения и определять их семантику в предписании для своих нужд.

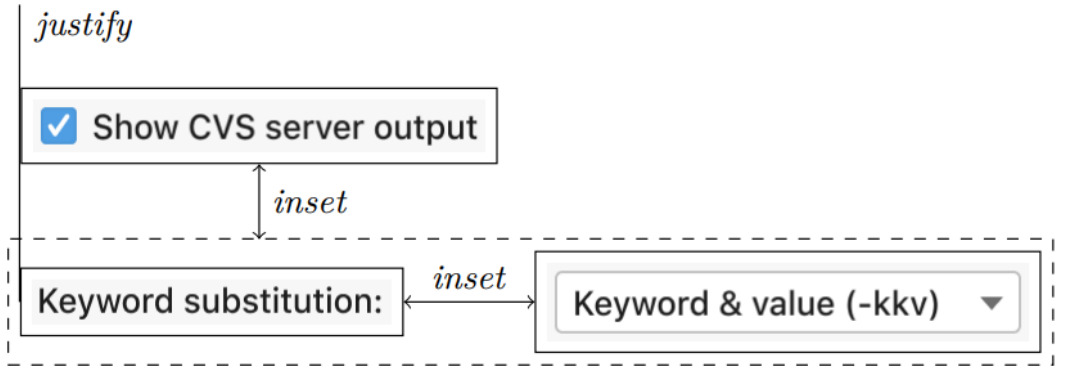


Рис. 3. Пример расположения элементов управления.
Pic. 3. Sample GUI form layout.

Для конкретной структуры её расположение (англ. layout) определяется набором примитивов расположения, выравниванием и другими подобными свойствами. В примере метка и поле со списком располагаются горизонтально с некоторым отступом, галочка располагается сверху от введённого виртуального элемента для пары метка-поле с другим отступом, и всё расположение выровнено влево, что изображено на рис. 3. В отличие от отношений, использованных при описании структуры, все примитивы расположения имеют фиксированную семантику, на которой основан процесс синтеза.

Наконец, предписание состоит из набора правил, описывающих какие примитивы расположения должны использоваться для соответствующих частей структуры и при каких условиях. На рис. 4 можно увидеть пример такого правила, взятого из правил [9]

проектирования интерфейсов компании JetBrains. Обычно, эти правила описываются неформально на примерах, в следующем разделе мы их формализуем.

09 If there are two input controls with labels of similar length that are separated from each other by a single control, align their input boxes on the left side.

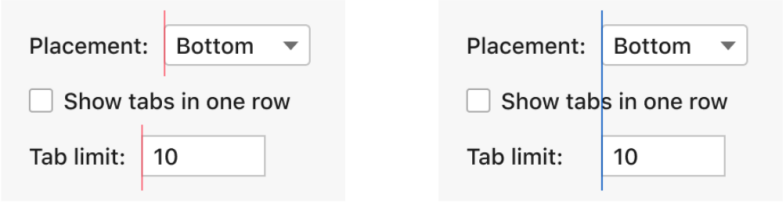


Рис. 4. Пример правила предписания.
Pic. 4. Sample GUI guideline rule.

3. Синтез расположения как задача удовлетворения ограничений

В нашей модели логическая структура, упомянутая в предыдущем разделе, описывается как набор отношений на некотором множестве элементов управления C. Обозначим нашу структуру $\Sigma = \{r_i^{n_i} \mid r_i^{n_i} \subseteq C^{n_i}\}$ как множество отношений, где $r_i^{n_i}$ является n_i-арным отношением над C; без потери общности мы не будем рассматривать такие отношения, как размер, текстовое содержимое элементов управления и т.п.

Реляционным шаблоном будем называть синтаксическую форму $r^n(X_1, \dots, X_n)$, где r^n – это n-арное отношение и X_j – это необязательно различные переменные.

Шаблоном расположения назовем синтаксическую форму $l(X_1, \dots, X_n)$, где l – это примитивы расположения, а X_j – переменные.

Как $s[X_i \leftarrow c_i]$ мы будем обозначать результат подстановки элементов управления $c_i \in C$ вместо переменных X_i в реляционный шаблон или в шаблон расположения s. Пусть $FV(s)$ – это множество всех переменных в реляционном шаблоне или в шаблоне расположения s, и будем называть *экземпляром расположения* шаблон без переменных.

Пусть $r^n(X_1, \dots, X_n)$ – это реляционный шаблон. Если для некоторого набора элементов управления $c_1, \dots, c_n \in C$ верно $(c_1, \dots, c_n) \in r^n$, то будем говорить, что результат подстановки $r^n(c_1, \dots, c_n)[X_i \leftarrow c_i]$ *встречается* в структуре Σ.

Правило (предписания) будем записывать как

$$p_1, \dots, p_k \mapsto l_1, \dots, l_m,$$

где p_i – реляционные шаблоны, l_j – шаблоны расположения. Также потребуем, чтобы любая переменная в правой части хотя бы раз встречалась в левой. Неформально, правило говорит, что если некоторые элементы управления некоторым образом соотносятся в структуре, то для них должны использоваться определенные примитивы расположения.

Предположим, нам дана структура Σ и множество экземпляров расположения S. Будем говорить, что это множество *подтверждается* набором правил R тогда и только тогда, когда для каждого элемента управления $l^* \in S$

(1) существует правило $p_1, \dots, p_n \mapsto l_1, \dots, l_i \dots, l_m$ в R, такое что $l^* = l_i[X_j \leftarrow c_j]$ для некоторых $\{c_j\} \subseteq C$, где $FV(l_i) = \{X_j\}$;

(2) для всех переменных $\{Y_1, \dots, Y_t\} = FV(p_1, \dots, p_n) \setminus FV(l_i)$ существуют элементы управления $c'_1, \dots, c'_t \in C$ такие, что для всех j подстановка $p_j[X_r \leftarrow c_r, \dots, Y_s \leftarrow c'_s]$ встречается в структуре Σ;

(3) $l_j[X_r \leftarrow c_r, \ldots, Y_s \leftarrow c'_s] \in S$ для всех j.

Неформально выражаясь, R подтверждает множество S тогда и только тогда, когда все элементы управления l^* могут быть корректно выведены из правил R. Это означает, что должно быть какое-то правило с шаблоном расположения l_i, который превратится в l^* с помощью некоторой подстановки переменных X_i. Но в этом правиле кроме X_i могут быть другие переменные Y_j. Если и для них существует такая подставка, которая совместно с подстановкой переменных X_i, заставит все реляционные шаблоны слева встречаться в Σ, тогда правило может быть применено и l^* выведено. К тому же, в правой части могут быть другие шаблоны расположения, и если и они окажутся тоже выведенными, то этот экземпляр расположения должен быть добавлен в S.

Подтверждённые множества содержат в себе *все* расположения, которые могут быть обоснованы правилами. Эти множества не содержат лишних расположений. Однако подтверждённое расположение элементов управления не делает его автоматически “хорошим” с точки зрения дизайнера. Пустое множество всегда подтверждается, но едва ли его можно назвать полезным расположением элементов. Из-за этого нам необходимо ввести ещё одно понятие: покрытие.

Будем говорить, что экземпляры расположения S *покрывают* структуру Σ тогда и только тогда, когда для каждого элемента управления $c \in C$ существует экземпляр расположения в S, который связывает c, т.е. содержит c как параметр. Покрытие неформально обозначает, что ни один элемент управления из структуры не остался не упомянутым в данном расположении.

Множество подтверждённых расположений в некотором смысле соответствует всем *корректным* расположениям с учетом правил, а покрытие – полноте. Однако мы воздержимся от введения этих понятий здесь, чтобы потом их употреблять в более общепринятом смысле.

Следующее требование навязано тем, что правила (предписания) должны применяться, а не игнорироваться. Будем называть подтвержденное и покрытое множество S *максимальным* тогда и только тогда, когда не существует подтвержденного и покрытого множества S', что $S \subset S'$.

Наконец, расположения могут содержать конфликты. Например, если элемент A должен быть расположен сверху B и наоборот, то, очевидно, что мы получили несовместные ограничения. Среди всех конфликтов мы можем выделить следующие:

- Не должно быть циклов, составленных только из примитивов расположения для вертикальной (горизонтальной) композиции.
- Не более одного элемента управления должно располагаться строго справа/снизу от другого элемента управления. Это и предыдущее требование проистекает из того, что отношения, составленные из этих примитивов, должны быть линейными порядками.
- Если два виртуальных элемента управления расположены горизонтально, то никакие два элемента управления внутри одного и другого не должны располагаться вертикально. Аналогично для вертикально расположенных виртуальных элементов управления.
- Никакие два элемента управления не могут быть расположены одновременно и вертикально, и горизонтально относительно друг друга.

Эти конфликты возникают из-за того, что мы располагаем элементы управления на двумерной плоскости (холсте). Кроме них бывают и другие конфликты, которые мы опустили для краткости изложения. Будем называть множества экземпляров без конфликтов *совместными*.

Итого, нам интересны подтвержденные, покрывающие структуру и совместные расположения с учетом выданного набора правил. Будем называть такие расположения *корректными*.

Стоит обсудить вопрос производительности подхода в худшем случае. Легко заметить, что нахождение корректного расположения – в общем случае NP-полная задача. Для подтверждения этого, рассмотрим решение NP-полной задачи поиска Гамильтонова пути в графе с помощью нашего подхода. Для произвольного ненаправленного графа мы можем построить структуру, в которой один элемент управления соответствует одной вершине графа, а некоторое бинарное отношение "E" – рёбрам. Затем рассмотрим такие правила расположения:

$$E\ (x, y) \mapsto \mathbf{hor}\ (x, y),$$
$$E\ (x, y) \mapsto \mathbf{hor}\ (y, x).$$

Любое корректное расположение с учётом правил по определению содержит все элементы управления (т.е. узлы графа) и только примитивы расположения вида "**hor**", где каждое вхождение примитива соответствует одному ребру в графе. Так как примитивы не могут образовывать цикл, и каждый элемент управления может быть связан ребром только с одним другим элементом, то ребра образуют Гамильтонов путь.

С другой стороны, можно заметить, что количество всех корректных расположений в худшем случае экспоненциально зависит от размера структуры. И выдача всех корректных расположений с помощью решателя, заставит его работать в худшем случае экспоненциально.

4. Реляционное программирование, верификаторы и решатели

Реализация синтеза интерфейсов использует реляционное программирование. В этом разделе мы кратко расскажем, что это такое, и как работает наш инструмент.

Реляционное программирование [5] – это подход, основанный на идее описания программ как отношений. Его можно рассматривать как вид логического программирования, в котором порицается использование всех не реляционных конструкций (эффекты, не логические конструкции). В узком смысле реляционное программирование – это написание программ на miniKanren – специально сконструированном для этих целей, встраиваемом предметно-ориентированном языке. В оригинале он разрабатывался для Scheme/Racket, позже miniKanren был портирован на другие языки[1]. Мы используем строго статически типизированную реализацию miniKanren, встроенную в OCaml, которая называется OCanren [10]. В miniKanren используется та же теория дизъюнктов Хорна, как и в Prolog, но с другим синтаксисом: явные унификации и дизъюнкции, конъюнкции, явное создание свежих (англ. fresh) переменных, а также используется стратегия *поиска с чередованием* (англ. interleaving search) [11], которая полна [12]. Кроме унификации с проверкой цикличности вхождений (англ. occurs-check) по умолчанию, miniKanren можно оснастить другими ограничениями, например: неравенства [13], конечные домены [14] или конструкции из номинальной логики [15].

В контексте нашей работы, самым важным свойством miniKanren является возможность выражения *обратимых вычислений*. Известно [16-17], что некоторые сложные программы могут быть построены как результат обращения некоторых других более простых программ. В частности, *решатель* задачи поиска можно рассматривать как обращение *верификатора* – программы, которая проверяет корректность ответа. Широко известно, что задача проверки корректности решения, как правило, гораздо проще задачи поиска корректного решения. Реляционная природа miniKanren позволяет осуществлять обратимые вычисления очень просто, что помогает в задачах синтеза программ [18–20].

[1] http://minikanren.org/#implementations

Другим компонентом нашего подхода является *реляционное преобразование* (англ. relational conversion). Во многих случаях (но не всегда) проще получить реляционную спецификацию из кода на функциональном языке, чем написать спецификацию вручную. Мы используем инструмент noCanren, который преобразует программы, написанные на подмножестве OCaml, в OCanren-спецификации, корректные по построению.

Продемонстрируем наш подход на следующем обозримом примере. Рассмотрим программу на рис. 5, которая складывает два натуральных числа в форме Пеано. Её реляционный образ, полученный с помощью реляционного преобразования (не буквально, но в эквивалентной записи) – на рис. 6. Сравнивая их, можно обратить внимание, что сопоставление с образцом было заменено на дизъюнкцию ($\vee$) и унификацию ($\equiv$); модификации потока данных – на конъюнкцию ($\wedge$); свежие переменные были добавлены, где это необходимо; а постфиксная нотация "^{o}" традиционно используется для обозначения определения реляций (вычислимых отношений). В отличие от функциональной реализации, у реляционной три аргумента (`x`, `y` и `z`) каждый из которых может содержать свежие переменные. Вычисление конструкции **`run`*** `{add`^{o} `x y z}` возвращает ленивый поток ответов, который содержит все подстановки этих трёх переменных, такие, что отношение $\{(x, y, z) \in \mathbb{N} \mid x + y = z\}$ верно. Этот поток можно исследовать в OCaml, чтобы получать ответы по одному. Одна и та же спецификация может использоваться и для сложения, и для вычитания, и для разбиения числа на два слагаемых.

```
let rec add x y =
  match x with
  | O → y
  | S xp → S (add xp y)
```

Рис. 5. Функциональная реализация сложения чисел Пеано.
Pic. 5. Functional implementation of Peano numbers addition.

```
let rec addᵒ x y z = ocanren {
  x ≡ O ∧ z ≡ y ∨
  fresh xp, zp in
    x ≡ S xp ∧
    z ≡ S zp ∧
    addᵒ xp y zp
}
```

Рис. 6. Реляционная реализация сложения чисел Пеано.
Pic. 6. Relational implementation of Peano numbers addition.

5. Синтез расположения GUI

В данном разделе мы сконструируем синтезатор расположений на основе предписания. Сделаем это за несколько шагов: начнём с простого функционального верификатора, сконвертируем его в реляционную форму, запустим в обратном направлении, проанализируем результат и применим некоторые оптимизации.

5.1 Функциональный верификатор

Устройство начального функционального верификатора на OCaml довольно прямолинейно. Воспользуется обычными для функционального программирования списками и алгебраическими типами данных, чтобы описать структуру и примитивы расположения. По

правилам предписания мы *синтезируем* верификатор, который в качестве аргументов принимает структуру и экземпляры расположения, и проводит прямолинейную проверку покрытия, совместности и подтверждения свойств, согласно определениям из раздела 3.

Покрытие можно легко получить, обойдя все примитивы расположения, собирая все невиртуальные элементы управления в множество, и проверяя, что оно соответствует множеству всех невиртуальных в структуре. Совместность проверяется аналогично, по определению.

Для подтверждения нам нужно обойти все примитивы расположения и подтвердить каждый их них. Это требует обращения правила предписания. Например, рассмотрим следующее правило:

$$\textbf{describes}\,(X,Y)\ \mapsto \textbf{vert}\,(X,Y),\ \textbf{halign}\,(X,Y).$$

$$\begin{aligned}
&\texttt{if}\ \ \textbf{vert}\,(X,Y)\in S\\
&\texttt{then}\ \ \textbf{describes}\,(X,Y)\in\Sigma\ \wedge\ \textbf{halign}\,(X,Y)\in S\\
&\texttt{else if}\ \ \textbf{halign}\,(X,Y)\in S\\
&\texttt{then}\ \ \textbf{describes}\,(X,Y)\in\Sigma\ \wedge\ \textbf{vert}\,(X,Y)\in S\,.
\end{aligned}$$

Оно определяет следующие случаи для процедуры подтверждения структуры Σ и набора экземпляров расположения S.

Мы *не проверяем максимальность* на данном шаге. Причина этому в особенности функционально-реляционного программирования. Чтобы проверить максимальность, нам нужно найти все остальные совместные экземпляры, которые и подтверждены, и покрыты. Но именно этой задачи мы хотим *избежать*, написав функциональный верификатор. Поэтому, реализация этого на функциональном языке подрывает всю идею использования реляционного программирования. Подчеркиваем, что это довольно частый случай для нашего подхода: для удобства программирования нам необходимо аккуратно провести границу между функциональным и реляционным кодом, чтобы не делать то, что мы получим позже за просто так.

Максимальность проверяется другим образом. Мы применяем реляционное преобразование для функционального верификатора, и запускаемся на конкретной структуре и на *свободной переменной* на месте конкретных экземпляров расположения. За счет полноты поиска miniKanren выдаст нам все множества, одновременно совместные, покрытые и подтверждённые. Затем мы просто отфильтруем не максимальные.

На этом первый шаг закончен, и мы получили полный неэффективный синтезатор. У наивной реализации присутствуют следующие недостатки:

- Синтезатор выдает (экспоненциально) большое количество эквивалентных ответов. Мы использовали списки для представления множеств, и синтезатор выдал нам все перестановки как уникальные ответы.
- Мы узнали, что представление структуры с помощью типов данных чрезмерно, и мы можем напрямую строить реляции miniKanren по структуре.
- Синтезатор выдает много частичных (не максимальных) ответов, что замедляет синтез и делает его непрактичным.

В следующих подразделах мы решим эти проблемы.

5.2 Представление с помощью бинарных гиперкубов

Для первой проблемы мы применили специальное представление расположения (*бинарный гиперкуб*), в котором любое расположение представляется однозначно. В сущности, это побитовое представление множеств и отношений, которое по-разному определяется для

каждой структуры. Например, пусть у нас два элемента управления a и b, и, для простоты, два примитива расположения **vert** и **hor**. Следующие объявления типов на языке OCaml определяют представление расположений:

```
type rels s  = {vert : bool; hor : bool}
type α roles = {a : α; b : α}
type layout  = rels roles roles
```

Этот подход нивелирует накладные расходы на поиск эквивалентных ответов с разными представлениями. Однако он требует специализации функционального верификатора под конкретную структуру. Это не такое уж и большое неудобство, к тому же, улучшения из следующего подраздела, потребуют сходной специализации.

5.3 Представление структуры как набора реляционных отношений

Как уже известно, структура – это набор отношений над элементами управления. В реляционном языке мы можем представлять отношения непосредственно как отношения. Предположим, мы работаем с фрагментом структуры с рис. 7. В оригинальном изложении она представлялась бы как данные, используя соответствующие типы для элементов управления и отношений. Однако, те же самые части структуры можно закодировать непосредственно в OCanren (рис. 8).

type $(A, \text{checkbox})$
type (B, label)
describes (B, A)

Рис. 7. Отношения структуры в абстрактной записи.
Pic. 7. Structure relations in abstract form.

```
let typeᵒ x y = ocanren {
  y ≡ Checkbox ∧ x ≡ A ∨
  y ≡ Label ∧ x ≡ B
}
let describesᵒ x y = ocanren {
  x ≡ B ∧ y ≡ A
}
```

Рис. 8. Реляционное представление отношений структуры.
Pic. 8. A relational representation of the structure relations.

После реляционного преобразования, шаблоны над структурой будут преобразованы в обычную реляционную форму. Это преобразование полностью убирает накладные расходы на интерпретацию для сопоставления структуры. Недостатком является то, что нам необходимо порождать эту часть системы для каждого изменения в структуре. Это естественный компромисс для подходов на основе специализации.

5.4 Жадный алгоритм разрешения конфликтов

Изначальная реализация порождала все возможные совместные множества экземпляров расположения. Но их число огромно, даже если мы рассматриваем только подтверждённые и покрытые. Чтобы сделать синтезатор применимым, нужно устранить эту неэффективность. Воспользуемся жадным подходом: вместо порождения всех подходящих и последующей

фильтрации, построим максимальное несовместное расположение, а затем найдем все конфликты и устраним их путём *недетерминированной* отмены минимального набора правил, которые образуют конфликты. Такой подход даст нам множество всех максимальных совместных расположений. Каждый шаг нами реализован как отдельная реляционная компонента.

На первом шаге мы недетерминировано применяем все правила предписания, чтобы получить максимальное (возможно, несовместное) множество экземпляров расположения, представленное гиперкубом.

На втором шаге воспользуемся особым реляционным верификатором, чтобы найти все конфликты. Для каждого вида конфликта применим специальный верификатор, который по гиперкубу и набору экземпляров расположения, проверяет, что

- множество содержится в гиперкубе;
- оно действительно содержит искомый конфликт.

Запустившись в обратном направлении на конкретном гиперкубе верификатор вернёт множество экземпляров расположения, которое образует конфликт соответствующего вида. Запуск в обратном направлении дизъюнкции верификаторов даст нам все конфликтующие экземпляры.

На третьем шаге мы для каждого конфликтующего расположения разрешаем конфликты, что даст нам (для каждого набора) множество максимальных совместных расположений. Сделаем это следующим образом. Обозначим множество конфликтующих экземпляров как S. Каждый найденный конфликт можно разрешить, убрав только один примитив расположения. Поэтому, будем по одному выкидывать экземпляры из S и смотреть какие конфликты разрешились. Заметим, что выкидывание одного примитива может разрешить сразу несколько конфликтов. Когда не останется ни одного конфликта, мы вернём максимальное совместное расположение. Однако, тут есть одна тонкость, осложняющая реализацию. Удаляя экземпляры, мы можем получить расположение, которое более не подтверждается предписанием, так как некоторые части расположения могут появляться и исчезать только одновременно. Поэтому, надо аккуратно реализовывать процедуру отмены правил. Весь шаг реализован реляционно, из-за многочисленного использования недетерминизма.

Обращаем внимание, что решение задействует три реляционные программы, которые передают результаты друг в друга. Эти программы нельзя объединить в одну, так как запуск следующей требует нахождения всех результатов предыдущей программы.

5.5 Вычисление абсолютных координат

Чтобы привести расположение к окончательному виду, необходимо посчитать координаты всех элементов управления. Для корректного множества экземпляров расположения эта задача сводится к решению задачи *в целочисленных линейных ограничениях*. Каждый примитив расположения добавляет некоторые неравенства в систему неравенств. Например, примитив $\mathbf{hor}\ (C_1, C_2)$ – горизонтальное расположение элементов управления C_1 и C_2 друг за другом – порождает следующие ограничения:

$$C_2.x - C_1.x \le i_H + C_1.width,$$
$$C_2.y - C_1.y = a_v,$$

где *переменные* $C_i.x$ и $C_i.y$ обозначают координаты x и y соответствующего элемента управления, целочисленные *константы* $C_i.width$ – ширину элемента C_i, i_H – горизонтальный отступ, a_v – отступ для вертикального выравнивания.

Также, для каждого виртуального элемента управления C, содержащего $C_1, \dots, C_n$, нужно добавить следующие ограничения на координаты, ширину и высоту:

$$C.x = \min\{C_1.x, \dots, C_n.x\},$$
$$C.y = \min\{C_1.y, \dots, C_n.y\},$$
$$C.x + C.width = \max\{C_i.x + C_i.width\}_{i=1}^{n},$$
$$C.y + C.height = \max\{C_i.y + C_i.height\}_{i=1}^{n}.$$

Дополнительно, нужно добавить неравенства, которые ограничивают максимальные значения переменных на основе размера холста, где мы располагаем элементы управления.

Существует множество способов разрешать линейные целочисленные неравенства, применение SMT решателя для теории линейной арифметики – один из них. Но заманчиво применить реляционный верификатор ещё раз. Этот решатель мог бы быть бесшовно интегрирован в существующую процедуру синтеза, что позволило бы проверять некоторые ограничения на более ранних фазах поиска.

Однако, текущий реляционный решатель показывает низкую производительность в присутствии виртуальных элементов управления. Как было упомянуто выше, размеры виртуальных элементов – не константы, что существенно увеличивает пространство поиска. Поэтому, на данный момент, мы используем Z3 [7], чтобы определять абсолютные координаты элементов управления, а также размеры виртуальных.

Заметим, что данный подход может повредить полноте решателя. Определение конфликтов, а также процедура их устранения из предыдущего раздела, никак не учитывают размеры холста и размеры виртуальных элементов (которые зависят только от размеров, входящих в них элементов). Таким образом, система целочисленных неравенств может быть несовместной даже для корректных расположений. Заметим, что отмена правил предписания влечет за собой отмену целочисленных ограничений, что может восстановить корректность системы. Другими словами, несовместность системы может быть решена также, как и конфликт – отменой правил. Отметим, что вся идея поиска конфликтов основана на предположении, что мы знаем заранее, что некоторые экземпляры расположения рано или поздно создадут несовместные ограничения на координаты. Но на стадии поиска и устранения конфликтов у нас недостаточно информации, чтобы решить эту задачу достаточно точно.

Восстановление полноты требует аккуратного взаимодействия с Z3. В случае невыполнимости системы S мы получаем *минимальное ядро невыполнимости* [21] (англ. unsatisfiable core). Это такое несовместное подмножество $UC \subseteq S$, что удаление одного ограничения из ядра делает систему выполнимой. На первый взгляд, достаточно недетерминировано удалять по одному ограничению из UC, и смотреть из каких частей расположения это ограничение появилось. Отменяя правило предписания, которое внесло это ограничение, мы сделаем систему выполнимой. К сожалению, не со всеми ограничениями можно так обойтись. Разделим UC на ограничения расположения L (порождённые экземплярами), и ограничения на размер Z: появившиеся из-за размера холста и размеров виртуальных элементов. Мы можем непосредственно отменять ограничения из L, но если не только они образуют ядро невыполнимости, то необходимо действовать более тонко.

Построим множество M, выбирая по одному ограничения c из L и запрашивая у Z3 решение системы $S \setminus \{c\}$ на каждом шаге. В итоге может получиться два результата:

(1) Множество $S \setminus \{c\}$ остается невыполнимым. Это означает, что c не влияет на невыполнимость и поиск продолжается.

(2) Множество $S \setminus \{c\}$ стало выполнимым. Значит c в конфликте с каким-то другим ограничением. Добавляем c в M и продолжаем дальше.

После окончания выполнения процедуры множество M будет содержать по построению все ограничения, такие что удаление одного из них восстанавливает выполнимость. Каждое из них появилось из-за какого-то экземпляра расположения. Будем недетерминировано отменять применение правил предписания, которые создали эти экземпляры. Из S мы

получим n подсистем, где $n = |M|$. Затем попробуем решить эти системы с помощью Z3 и для каждой повторим процедуру. В конце концов мы либо отменим достаточно правил, чтобы получить все корректные расположения, либо придём к состоянию, где система невыполнима, а ядро содержит только ограничения на размер, что означает, что элементы не помещаются на наш холст, как их не раскладывай. Это обосновывает полноту алгоритма синтеза. Псевдокод процедуры представлен на рис. 9.

```
solve (lay_part, size_part) {
    // Try to solve the full system
    if (Z3.solve (lay_part ∪ size_part)) {
        // Return a model
        return [ Z3.model ]
    }

    // If unsat we need to find and solve layout conflict
    // MUC is one of conflicts
    muc = Z3.minimal_unsat_core
    // A list of refined models
    answers = []

    for (muc_element ∈ muc) {
        // To solve the conflict we need to remove any layout
        // element of MUC from the system
        if (muc_element ∈ lay_part) {
            reduced_lay_part = lay_part \ {muc_element}
            answers ++= solve (reduced_lay_part, size_part)
        }
    }

    return answers
}
```

Рис. 9. Алгоритм решения конфликтов координат.
Pic. 9. Solving coordinate conflicts algorithm.

6. Реализация и эксперименты

Прототип синтезатора реализован как клиент-серверное приложение[2]. Со стороны клиента мы используем HTML5 и JavaScript для взаимодействия с сервером. На сервере исполняется OCaml приложение, оснащённое OCanren и Z3.

Чтобы облегчить пользователю представление структуры GUI и предписания, мы разработали два текстовых предметно-ориентированных языка. В реализации выразительность языка предписаний расширена: мы разрешаем в правилах использовать пользовательские ограничения, присутствуют конструкции для задания табличных расположений, и т.п. Ни одно из расширений существенно не изменяет подход, описанный в предыдущих главах, и все расширения могут быть учтены. Мы также параметризуем систему набором констант, которые определяют свойства окружения (например, размер холста, где мы располагаем элементы управления). Пример описания структуры для диалогового окна настроек поиска сервиса Google Drive представлена на рис. 10. Текстовое описание предписания слишком длинное, чтобы представить его здесь.

Для оценки нашего прототипа на промышленных примерах мы также реализовали отображение элементов управления с помощью Qt/QML. На рис. 11 можно увидеть два

[2] Прототип доступен по ссылке: https://se.math.spbu.ru/projects/genui (проверено: 26.08.2024).

расположения структуры диалогового окна Google Drive, синтезированного с учетом двух немного различных версий правил проектирования интерфейсов от JetBrains [9] (по которым мы составили два различных предписания), нарисованные с помощью QtQuick Controls[3] и темы оформления Material Design. Предписание регламентирует, что “слишком длинный” текст должен располагаться под меткой. На правом расположении константа, регламентирующая свойство “слишком длинный” была уменьшена. Синтез занял примерно 5 секунд. Некоторые другие примеры можно найти на рис. 12.

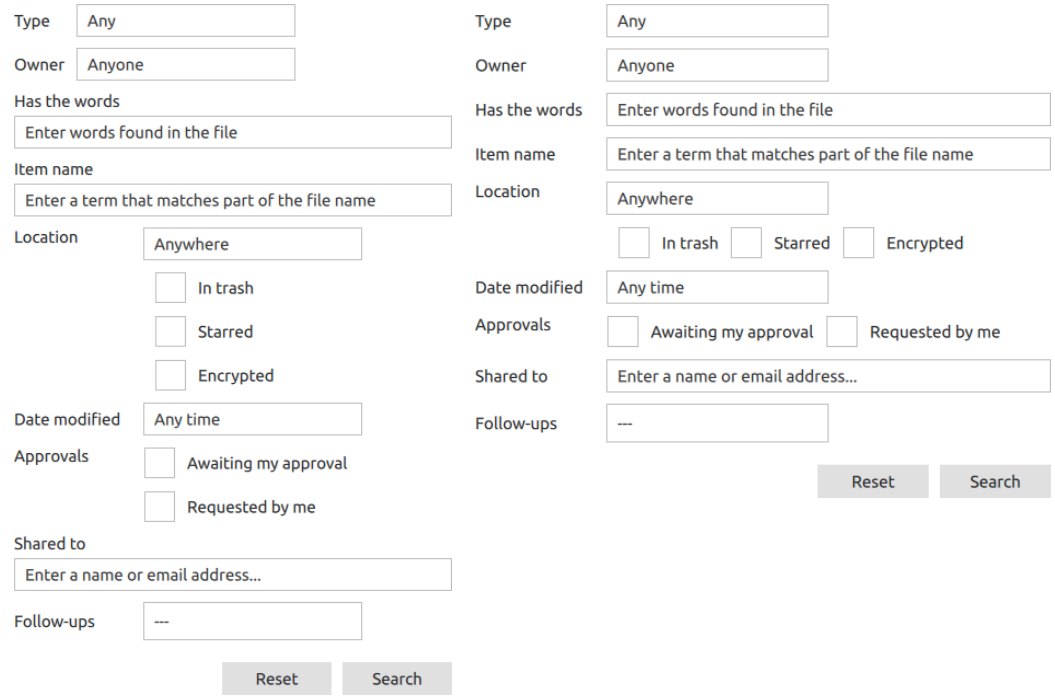
7. Обзор

Дизайн и реализация GUI – горячая тема уже много десятилетий, поэтому можно найти много инструментов, подходов, статей и отчётов по теме. Большая часть из них (если не все) предлагают декларативные и автоматические решения. Но если присмотреться, то их понятия “декларативности” и “автоматизации” отчаются от наших.

```
ordered main_group (
  Label "Type" describes TextEdit "Any" {width=200}
  Label "Owner" describes TextEdit "Anyone" {width=200}
  Label "Has the words" describes
    TextEdit "Enter words found in the file" {width=400}
  Label "Item name" describes
    TextEdit "Enter a term that matches part of the file name" {
      width = 400
    }
  (Label "Location" describes
    (TextEdit "Anywhere" { width = 200 }) dominates
    ordered location_checkboxes (
      Label "In trash" describes CheckBox in_trash_checkbox
      Label "Starred" describes CheckBox starred_checkbox
      Label "Encrypted" describes CheckBox encrypted
    )
  )
  Label "Date modified" describes TextEdit "Any time" {
    width=200
  }
  Label "Approvals" describes
    ordered approvals_checkboxes (
      Label "Awaiting my approval" describes
        CheckBox awaiting_checkbox
      Label "Requested by me" describes
        CheckBox requestred_checkbox
    )
  Label "Shared to" describes
    TextEdit "Enter a name or email address..." {width=400}
  Label "Follow-ups" describes
    TextEdit "---" {width=200, height=35}
)
Button "Search" approves^ main_group
Button "Reset" cancels^ main_group
```

Рис. 10. Описание структуры диалогового окна настроек поиска из сервиса Google Drive.
Pic. 10. Google Drive search settings dialog structure description.

[3] https://doc.qt.io/qt-5/qtquick-controls2-qmlmodule.html (проверено: 26.08.2024).



(a) Если метка описывает "слишком длинный" элемент управления, то располагать их вертикально
(a) If label describes controls which are "too long", then place them vertically

(b) То же самое, но константа для свойства "слишком длинный" уменьшена
(b) The same, but the constant for "too long" was decreased

Рис. 11. Синтезированные расположения элементов для диалогового окна из Google Drive.
Pic. 11. Google Drive search settings dialog layouts.

Во-первых, необходимо упомянуть некоторое количество инструментов для реализации GUI и визуализации данных, например, React [22], Jetpack Compose [23], SwiftUI [24], Streamlit [25], D3 [26] и другие. Они предоставляют пользователю наборы примитивов, которые позволяют отображать данные и пользовательский интерфейс. Например, Streamlit предоставляет набор встроенных примитивов отображения: "колонки", "контейнеры", "модальные диалоги" и т.п. [27], а также множество внешних компонент. Эти примитивы позволяют пользователям абстрагироваться от конкретного вычисления координат и относительного выравнивания. Также они предоставляют разумное поведение при изменении размера холста. Однако, выбор конкретного примитива остаётся на усмотрение разработчика, а не определяется системой. И если расположение элементов должно по какой-то причине измениться, то эти изменения нужно реализовывать вручную. В нашем случае, разработчики не задают конкретное расположение, а только логическую структуру пользовательского интерфейса. Правила проектирования интерфейсов определяют конкретное расположение элементов управления, в зависимости от внешних ограничений, таких как разрешение экрана или настройки региона (например, письмо справа-налево). Пока логическая структура не изменяется, не потребуется никакого взаимодействия с программистом для отображения интерфейса по-другому. С другой стороны, данные инструменты могут использоваться совместно с нашим как способ визуализации интерфейса, так как они предоставляют сходные примитивы расположения. В этой работе мы таким образом задействовали Qt/QML. Программирование в ограничениях уже использовалось для

размещения элементов управления GUI. Одним из примеров реактивного языка программирования в ограничениях является язык Wallingford [28] и система Cassowary [29]. Wallingford позволяет прикреплять ограничения различной силы к значениям в программе. Система реагирует на изменения времени и обновляет значения, не нарушая ограничения. Например, можно реализовать элемент управления с шириной равной синусу текущего времени. Система Cassowary и её потомки позволяют вычислять размеры и положения элементов динамически, например, при изменении размера холста. Она поддерживает множество ограничений, в том числе глубину по оси Z; арифметические операции (например, ширина одного элемента может быть половиной высоты другого); элементы, накладывающиеся друг на друга и т.д. Эта система предназначена для задач автоматической изменения размеров элементов при изменении общего размера холста. Также, она не предоставляет поддержки правил общего вида для корректного позиционирования элементов. Мы сомневаемся в выразимости неоднозначных расположений элементов в данных системах, например, если вертикально или горизонтальное взаимоположение определяется на основе размеров. С другой стороны, количество различных ограничений больше, чем у нас. Например, поддержка накладывающихся друг на друга элементов управления запрещена у нас с самого начала, и такие расположения не создаются вообще.

Рис. 12. Примеры структур (слева) и синтезированных расположений (справа) с учётом правил проектирования интерфейсов от JetBrains.
Pic. 12. Examples of structures (left) and synthesized layouts (right) w.r.t. JetBrains guidelines.

В последние годы появились методы на основе машинного обучения. Некоторые работы ставят гораздо более амбициозные цели, чем наши.

Существует направление исследования, занимающееся порождением кода UI на основе картинок [30]. По полученному от дизайнера рисунку интерфейса, инструмент распознает

элементы управления и их относительное местоположение, и создает код реализации для инструмента создания пользовательских интерфейсов. Подход ортогонален и не совместим с предлагаемым нами. Он требует взаимодействия с дизайнером во время получения каждой части интерфейса, у нас же дизайнер задействован только при описании предписания. Также наш подход позволяет создавать много интерфейсов автоматически. В инструменте [31] была поставлена несколько другая задача: по картинке интерфейса синтезируется его реализация в терминах примитивов Android GUI, масштабируемая между различными устройствами и без типичных ошибок. Для выполнения этой задачи из картинки извлекается элементы управления, их местоположения, а также некоторые ограничения. Эти ограничения напоминают наши примитивы расположения, но специализированы для семантики виджета `ConstraintLayout` [32] из Android. На основе этих ограничений и набору *свойств надёжности*, разработанному авторами, вероятностная модель, обученная на большем наборе существующих интерфейсов, порождает код. Эти реализации, более устойчивы к измерению размера и разрешения экрана, чем полученные только путём распознавания картинок. Любопытно, что авторы мотивируют свою работу, заявляя, что “одно и то же расположение должно отображаться более чем на 15000 устройствах Android с ≈100 различных плотностей пикселей; требовать от программистов разработки и поддержки всех сочетаний входа чрезвычайно нежелательно”. Именно это делает наша система за несколько минут со 100% точностью, поэтому мы считаем свой подход более общим.

Также решалась задача согласованного во всём приложении оформления GUI [33]. Подход основан на идее заполнения: в предположении, что уже есть набор согласованных расположений, решается проблема добавления ещё одного элемента так, чтобы новое расположение было согласовано со старым. Эта задача связана с нашей, так как мы можем рассматривать уже существующее расположение как неявно заданное предписание, но мы можем указать несколько потенциальных недостатков. Во-первых, учитывается только добавление элементов, без удаления. Во-вторых, добавление/удаление компонент не обязательно приводит к “монотонному” изменению расположения: добавление ещё одного поля ввода может существенно поменять расположение. Наконец, начальный согласованный дизайн редко появляется из воздуха, наверняка это результат следования уже существующему предписанию, которое было описано явно.

В системе [34] поставлена гораздо более широкая задача синтеза без предписания, а только на основе эстетических, эргономических и других метрик. Для синтеза используется генетический алгоритм, а качество вычисляется на основе отзывов пользователей. Сам синтез работает часами. Хотя подход потенциально позволяет создавать эстетически приятные интерфейсы без участия дизайнера, задача получения корректного расположения для существенно разных структур обходится стороной.

Встречается интересная задача [35] исследования различных расположений элементов. Она ставит своей целью помочь дизайнерам разрабатывать убедительные и разнообразные расположения элементов управления. Вводится набор ограничений, с помощью которого дизайнеры составляют требования к результату. Любопытно, что в наших терминах этот набор ограничений является смесью структурных ограничений и ограничений на расположение: например, можно указывать и порядок элементов, и выравнивание. По набору ограничений система порождает множество расположений, удовлетворяющих им. Изменяя ограничения дизайнер исследует возможные интерфейсы. Для решения ограничений используется метод ветвей и границ. С ростом числа возможных решений применяются эвристики для отсечения эстетически неподходящих ответов. Несмотря на то, что эта работа похожа на нашу, она нацелена на дизайнеров и может рассматриваться скорее, как средство разработки предписаний, чем как средство получения расположения элементов, удовлетворяющего предписанию.

8. Заключение

Данная работа посвящена синтезу GUI на основе предписания. Наш подход позволяет автоматически располагать элементы управления GUI так, что результат по построению соответствует набору правил, сформулированных дизайнером. Прототип реализации позволяет синтезировать реалистичные примеры промышленных интерфейсов, с учетом промышленных предписаний за разумное время.

Одним из интересных вопросов, является необходимость использования реляционного программирования для решения нашей задачи. Да, набор правил, регламентирующих систему переписывания, в принципе, может быть реализован без использования реляционного подхода. Однако мы считаем, что в этом случае большая часть работы по обоснованию корректности решения должна быть повторена заново. В нашем же случае обоснование тривиально следует из полноты поиска miniKanren и полноты по опровержению (англ. refutational completeness) нашего решения. Мы также предсказываем, что решение потребует изобретения заново недетерминизма и поиска с возвратами, которые уже есть в реляционном программировании. Также дуальность между экземплярами в структуре и реляционными отношениями, изначально не ожидаемая нами, по нашему мнению, демонстрирует, что реляционное программирование – это подходящий подход для решения поставленной задачи.

Список литературы / References

[1]. Haft M., Humm B., Siedersleben J. The Architect's Dilemma – Will Reference Architectures Help? Quality of Software Architectures and Software Quality, 2005, pp. 106-122.

[2]. Ingram S. (2016) The Thumb Zone: Designing for Mobile Users. Smashing Magazine (online). Available at: https://www.smashingmagazine.com/2016/09/the-thumb-zone-designing-for-mobile-users, accessed 30.08.2024.

[3]. Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems. ISO 9241-210:2019, International Organization for Standardization, 2019. Available at: https://www.iso.org/standard/77520.html.

[4]. Gerber E., Carroll M. The psychological experience of prototyping. Design Studies, vol. 33, isuue 1, 2012, pp 64-84. DOI: 10.1016/j.destud.2011.06.005.

[5]. Friedman D.P., William W.E., Kiselyov O., Hemann J. The Reasoned Schemer. The MIT Press, 2nd edition, Cambridge, USA, 2005, 224 p.

[6]. Lozov P., Verbitskaia E., Boulytchev D. Relational Interpreters for Search Problems. In miniKanren and Relational Programming Workshop, 2019.

[7]. Leonardo de Moura, Bjørner N. Z3: An Efficient SMT Solver. Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, 2008, pp. 337-340.

[8]. Bengfort J. Thin vs. Thick vs. Zero Client: What's the Right Fit for Your Business? Online. Available at: https://biztechmagazine.com/article/2018/10/thin-vs-thick-vs-zero-client-whats-right-fit-your-business-perfcon, accessed: 30.08.2024.

[9]. Intellij platform UI guidelines: Layout (online). JetBrains s.r.o., 2000-2022. Available at: https://jetbrains.github.io/ui/principles/layout, accessed: 30.08.2024.

[10]. Kosarev D., Boulytchev D. Typed Embedding of a Relational Language in OCaml. Electronic Proceedings in Theoretical Computer Science, 2016, pp. 1-22.

[11]. Kiselyov O., Chung-chieh Shan, Friedman D.P., Amr S. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, New York, USA, 2005, pp. 192-203.

[12]. Rozplokhas D., Vyatkin A., Boulytchev D. Certified Semantics for Relational Programming. Programming Languages and Systems, APLAS 2020, Lecture Notes in Computer Science, vol. 12470, Springer, Cham, pp. 167-185.

[13]. Comon H. Disunification: A Survey. Computational Logic – Essays in Honor of Alan Robinson. MIT Press, 1991, 322-359.

[14]. Alvis C.E., Willcock J.J., Carter K.M., Byrd W.E., Friedman D.P. cKanren: miniKanren with Constraints. Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming, 2011.

[15]. Byrd W.E., Friedman D.P. αKanren A Fresh Name in Nominal Logic Programming. In Scheme and Functional Programming, 2007.
[16]. Abramov S., Glück R. From Standard to Non-Standard Semantics by Semantics Modifiers. International Journal of Foundations of Computer Science, vol. 12, issue 2, 2001, pp. 171-211. DOI: 10.1142/S0129054101000448.
[17]. Abramov S., Glück R. Combining Semantics with Non-standard Interpreter Hierarchies. FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science, Springer Berlin Heidelberg, 2000, pp. 201-213.
[18]. Byrd W.E., Holk E., Friedman D.P. MiniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). Proceedings of the Annual Workshop on Scheme and Functional Programming, Association for Computing Machinery, New York, USA, 2012, pp. 8-29.
[19]. Byrd W.E., Ballantyne M., Rosenblatt G., Might M. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). Proceedings of ACM Program. Lang., Association for Computing Machinery, New York, USA, 2017, pp. 8:1-8:26.
[20]. Kosarev D., Lozov P., Boulytchev D. Relational Synthesis for Pattern Matching. Programming Languages and Systems, Springer International Publishing, Cham, 2020, pp.293-310.
[21]. Guthmann O., Strichman O., Trostanetski A. Minimal Unsatisfiable Core Extraction for SMT. 2016 Formal Methods in Computer-Aided Design (FMCAD), Mountain View, CA, USA, 2016, pp. 57-64. DOI: 10.1109/FMCAD.2016.7886661.
[22]. React: A JavaScript Library for Building User Interfaces. Meta Platforms, Inc. Available at: https://reactjs.org/, accessed: 30.08.2024.
[23]. Jetpack Compose. Android Developers. Available at: https://developer.android.com/jetpack/compose, accessed: 30.08.2024.
[24]. SwiftUI. Apple Inc. Available at: https://developer.apple.com/xcode/swiftui, accessed: 30.08.2024.
[25]. Streamlit framework site. Available at: https://docs.streamlit.io, accessed: 30.08.2024.
[26]. The JavaScript library for bespoke data visualization. Available at: https://d3js.org, accessed: 30.08.2024.
[27]. Streamlit layouts and containers. Available at: https://docs.streamlit.io/develop/api-reference/layout, accessed: 30.08.2024.
[28]. Borning A. Wallingford: Toward a Constraint Reactive Programming Language. Companion Proceedings of the 15th International Conference on Modularity, Association for Computing Machinery, New York, NY, USA, 2016, pp. 45-49. DOI: 10.1145/2892664.2892667.
[29]. Badros G.J., Borning A., Stuckey P.J. The Cassowary Linear Arithmetic Constraint Solving Algorithm. ACM Trans. Comput.-Hum. Interact., vol. 8, issue 4, Association for Computing Machinery, New York, NY, USA, 2001, pp. 267-306. DOI: 10.1145/504704.504705.
[30]. Bo Cai, Jian Luo, Zhen Feng. A novel code generator for graphical user interfaces. Scientific Reports, vol. 13, 2023. DOI: 10.1038/s41598-023-46500-6.
[31]. Bielik P., Fischer M., Vechev M. Robust relational layout synthesis from examples for Android. Proc. ACM Program. Lang., vol. 2, Association for Computing Machinery, New York, NY, USA, 2018. DOI: 10.1145/3276526.
[32]. Android ConstraintLayout widget. Accessed: 30.08.2024, available at: https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout.
[33]. Brückner L., Leiva L.A., Oulasvirta A. Learning GUI Completions with User-defined Constraints. ACM Trans. Interact. Intell. Syst., vol. 12, Association for Computing Machinery, New York, NY, USA, 2022. DOI: 10.1145/3490034.
[34]. Shiripour M., Dayama N.R., Oulasvirta A. Grid-based Genetic Operators for Graphical Layout Generation. Proc. ACM Hum.-Comput. Interact., vol. 5, Association for Computing Machinery, New York, NY, USA, 2021. DOI: 10.1145/3461730.
[35]. Swearngin A., Wang C., Oleson A., Fogarty J., Amy J. Ko. Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints. Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, 2020. Available at: https://api.semanticscholar.org/CorpusID:210177012, accessed: 30.08.2024.

Информация об авторах / Information about authors

Дмитрий Сергеевич КОСАРЕВ является ассистентом кафедры Системного программирования мат-мех. факультета СПбГУ. Его научные интересы включают функциональное программирование, компиляторы и реляционное программирование.

Dmitry Sergeevich KOSAREV is an assistant at the Chair of System Programming of Mathematics and Mechanics Faculty of SPBU. His research interests include functional programming, compilers, and relational programming.

Петр Алексеевич ЛОЗОВ является кандидатом физико-математических наук. Сфера научных интересов: статический анализ, трансляция языков программирования, функционально-реляционное программирование.

Petr Alekseevich LOZOV – Cand. Sci. (Phys.-Math.). Research interests: static analysis, translation of programming languages, functional-relational programming.

Дмитрий Юрьевич БУЛЫЧЕВ – доцент кафедры системного программирования мат.-мех. факультета СПбГУ, кандидат физико-математических наук. Область научных интересов - языки и инструменты программирования, компиляторы, функциональное, логическое и реляционное программирование, анализ и синтез программ.

Dmitry Yuryevich BOULYTCHEV – Cand. Sci. (Phys.-Math.), associate Professor of the System Programming Chair of the Faculty of Mathematics and Mechanics of SPBU. His research interests include programming languages and tools, compilers, functional, logical and relational programming, program analysis and synthesis.