Tracing Program Transformations with String Origins*

Pablo Inostroza[1], Tijs van der Storm[1,2], and Sebastian Erdweg[3]

[1] CWI, Amsterdam, The Netherlands [2] INRIA Lille, France [3] TU Darmstadt, Germany

Abstract. Program transformations play an important role in domain-specific languages and model-driven development. Tracing the execution of such transformations has well-known benefits for debugging, visualization and error reporting. In this paper, we introduce *string origins*, a lightweight, generic and portable technique to establish a tracing relation between the textual fragments in the input and output of a program transformation. We discuss the semantics and the implementation of string origins using the Rascal meta programming language as an example. We illustrate the utility of string origins by presenting data structures and operations for tracing generated code, implementing protected regions, performing name resolution and fixing inadvertent name capture in generated code.

1 Introduction

Program transformations play an important role in domain-specific language (DSL) engineering and model-driven development (MDD). In particular, DSL compilers are often structured as a sequence of transformations, starting with an input program and eventually generating code. It is well-known that origin tracking [16] and model traceability [1,8,12,13,14] provide valuable information for debugging, error reporting and visualization.

In this paper, we focus on traceability for transformations that generate (fragments of) text. We propose *string origins*, a lightweight technique that links each character in the generated text to its origin. A string either originates directly from the input model, occurs as a string literal in the transformation definition, or is synthesized by the transformation (e.g., by string concatenation or substitution). We represent string origins using a combination of unique resource identifiers (URIs) and offset and length values that identify specific text fragments in a resource. We propagate string origins through augmented versions of standard string operators, such that the propagation is fully transparent to transformation writers. In particular, parsing and unparsing retains string origins for text fragments that appear in the AST, such as variable names.

Through applications of string origins we further confirm the usefulness of model traceability by realizing generic solutions to common problems in program-transformation design. First, string origins allow us to link generated elements back to their origin. In Section 3.1, we show how this enables the construction of editors with embedded hyperlinks to inspect generated code. Second, we present an example of attaching additional

* This research was supported by the Netherlands Organisation for Scientific Research (NWO) Jacquard Grant "Next Generation Auditing: Data-Assurance as a service" (638.001.214).

information to generated code via string origins. Section 3.2 describes how this enables protected regions in generated code. Third, string origins can be interpreted as unique pointers that identify subterms. In Section 3.3, we use the origins of symbolic names (variables, type names, method names, etc.) to implement name resolution. Finally, string origins can be used to systematically replace fragments of the generated code that have the same origin. In Section 3.4, we show a generic solution for circumventing accidental variable capture (hygiene) by systematic renaming of generated names.

In Section 4, we discuss the implementation of string origins in the context of Rascal [9]. Overall, we found that string origins have a number of important benefits that can improve the design of program transformations and transformation engines:

- Totality: Unlike existing work in origin tracking and model traceability [12], string origins induce an origin relation which is total. That is, the origin relation maps every character in the output text of a transformation back to its origin.
- Portability: Since the origin relation is based on string values and string operations instead of inferred from transformation code, the structure or style of the transformation language is largely irrelevant. As a result, string origins are portable across transformation systems, transformation styles, and technological spaces. Even in the case of graphical modeling languages, embedded strings (e.g., names, labels, etc.) could be annotated with their location in the serialization format used to store such models.
- Universality: String origins are independent of the source or target language, since they only apply to the primitive type string. In particular, origin propagation is independent of the AST structure or meta model.
- Extensibility: String origins are automatically propagated as annotations of substrings. As such, string origins can serve as general carriers of additional, domain-specific information. Marking certain subsstrings as protected (Section 3.2) is an example of this.
- Non-invasiveness: Transformation languages that support string manipulation during program transformation can support string origins by modifying the internal representation of strings, without changing the programming interface of strings. The only visible change is at input boundaries where strings are constructed.

We have implemented string origins as an experimental feature of Rascal, a meta programming language for source code analysis and transformation [9]. The applications and example code of this paper have all been prototyped in Rascal. The full code of the examples can be found online at `https://github.com/cwi-swat/string-origins`.

2 String Origins

We illustrate the basic idea of string origins in Figure 1. The code in the middle shows a simple transformation which converts name and email address specifications to the VCARD format. Arrows and shading indicate the origin relation. The white-on-black substrings in the output are introduced by the transformation; their origins point to the string template in the transformation code in the middle. In contrast, the substrings with gray backgrounds (name and email) are copied over from the input to the output, and

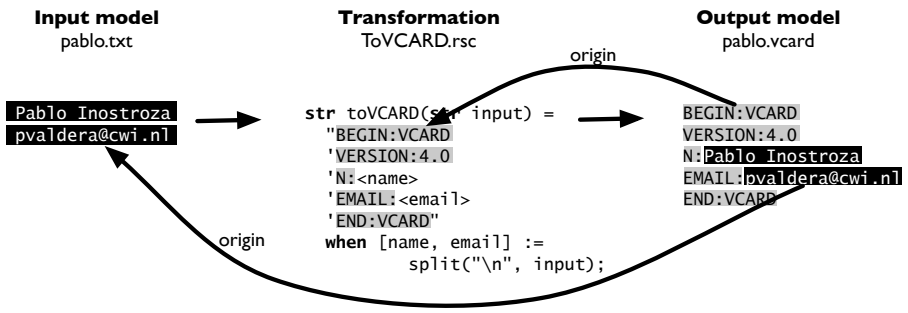


Fig. 1: Example of a simple Rascal transformation with trace links

hence point back to the input model. The substrings in the result are partitioned according to the origin relation: a fragment originates in either the input, or the transformation.

Note that the transformation processes the input by splitting the string. It is important to realize that this does not break the origin relation, but instead makes it more fine-grained: the output fragments "Pablo Inostroza" and "pvaldera@cwi.nl" have distinct origins pointers to the exact corresponding substrings in the input.

2.1 Representing string origins

Many transformations take text files as input and, eventually, produce text files as output. Moreover, the transformations themselves are expressed often as transformation code that is stored in text files as well. String origins exploit this fact by representing origins as *source locations*. Conceptually, a source location is a tuple consisting of a URI identifying a particular resource and an *area* identifying a text fragment within the resource. We represent an area by its start offset and length.

In the context of Rascal, source locations are represented by the built-in **loc** data type. To give an example, |file:///etc/passwd|(0, 50) identifies the first 50 characters in the file /etc/passwd, starting at offset 0. Rascal's source locations also represent begin and end line and column numbers, but for the remainder of this paper we will abstract from this technical detail. Although source locations are built into Rascal, they are easily implemented in any other transformation system.

The propagation of string origins is transparent: The transformation writer can fully ignore their presence and simply uses standard string operations such as concatenation or substitution. We discuss the details of the propagation in Section 4. Here, we want to highlight how to build generic tools on top of origin information. To this end, we provide an API for accessing locations and origins of substrings. First, we provide a function for decomposing a string into its atomic substrings (called chunks):

```
alias Index = rel[loc pos, str chunk];
Index index(str x, loc output);
```

Function index constructs an Index by collecting the atomic substrings of a string at a given location (e.g., a file path). The type Index is defined as a binary relation from the

location of a substring to the corresponding chunk. The relation type **rel** is native in Rascal and is equivalent to a set of tuples. Second, each of the chunks in an Index has an associated origin which can be retrieved with the function origin.

```
loc origin(str x); // require: x is a chunk
```

For example, we can call index on the generated VCARD shown in Fig. 1. Assuming the output location is |file:///pablo.vcard|, we get the following index:

```
{<|file:///pablo.vcard|(0,28),  "BEGIN:VCARD\nVERSION:4.0\nN:">,
 <|file:///pablo.vcard|(28,14), "Pablo Inostroza">,
 <|file:///pablo.vcard|(42,7),  "\nEMAIL:">,
 <|file:///pablo.vcard|(49,14), "pvaldera@cwi.nl">,
 <|file:///pablo.vcard|(63,9),  "\nEND:VCARD">}
```

Applying the origin function on any of the chunks retrieves the location where that particular chunk of text was introduced. Combining both functions gives us the origin relation, modeled by the Trace data type, which relates output locations to their corresponding origins:

```
alias Trace = rel[loc pos, loc org];
Trace trace(str s, loc out) = {<l, origin(chunk)> | <l, chunk> ← index(s, out)}
```

Function trace maps function origin over all chunks of the index. Considering again the example of Fig. 1, the trace relation of the generated VCARD looks as follows:

```
{<|file:///pablo.vcard|(0,28),  |file:///ToVCARD.rsc|(28, 28)>,
 <|file:///pablo.vcard|(28,14), |file:///pablo.txt|(0,14)>,
 <|file:///pablo.vcard|(42,7),  |file:///ToVCARD.rsc|(66, 7)>,
 <|file:///pablo.vcard|(49,14), |file:///pablo.txt|(15,14)>,
 <|file:///pablo.vcard|(63,9),  |file:///ToVCARD.rsc|(86, 9)>}
```

Note that the URIs in the origins distinguishes chunks originating in the input (pablo.txt) from chunks introduced by the transformation (ToVCARD.rsc). Both the index and trace relations are the stepping stones for the generic tools developed in the subsequent section.

2.2 String origins in M2T and M2M transformations

The previous example illustrates the use of string origins for text-to-text transformations. However, string origins are also useful in model-to-text and model-to-model transformations. More specifically, when parsing text into an AST, the string fragments that appear as leaves of the AST have string origins attached, pointing to the corresponding text fragment in the input file. Model-to-model transformations preserve the origins of strings copied from the input model and generate new origins for synthesized string fragments. Similarly, unparsing and other model-to-text transformations preserve the origins of strings in the AST. Again, the origin propagation is transparent to transformation writers, parsing and unparsing because origins are propagated through standard string operators.

Tracing origin information for string fragments in an AST is often useful. For example, variable names typically occur as string fragments in an AST. Figure 2 illustrates tracing of variable names in the context of a DSL for state machines. Figure 2a shows the source code of a state machine. Parsing the state machine produces an abstract

```
state opened
  close => closed
end
state closed
  open => opened
  lock => locked
end
state locked
  unlock => closed
end
```

(a) An example state machine

```
controller(
  [ ... /* event declarations */ ... ],
  [state("opened"@{|input|(62,6)}, [],
     [transition("close"@{|input|(70,5)},
                 "closed"@{|input|(79,6)})]),
   state("closed"@{|input|(100,6)},[],
     [transition("open"@{|input|(108,4)},
                 "opened"@{|input|(116,6)}),
      transition("lock"@{|input|(124,4)},
                 "locked"@{|input|(132,6)})]),
   state("locked"@{|input|(152,6)},[],
     [transition("unlock"@{|input|(160,6)},
                 "closed"@{|input|(170,6)})])])
```

(b) Parsed AST of the state machine

```
prog([
  fdef("opened"@{|input|(62,6)},[],val(nat(0))),
  fdef("closed"@{|input|(100,6)},[],val(nat(1))),
  fdef("locked"@{|input|(152,6)},[],val(nat(2))),
  ... // dispatch functions per state
  fdef( // main dispatch
    "main"@{|meta|(1280,13)},
    ["state"@{|meta|(1307,5)},
     "event"@{|meta|(1316,5)}],
    cond(equ(var("state"@{|meta|(1515,5)}),
             call("opened"@{|input|(62,6)}, [])),
      call("opened-dispatch"
           @{|meta|(1565,9),|input|(62,6)},
           [var("event"@{|meta|(1583,5)})]),
      cond(equ(var("state"@{|meta|(1515,5)}),
               call("closed"@{|input|(100,6)},[])),
        call("closed-dispatch"
             @{|input|(100,6),|meta|(1565,9)},
             [var("event"@{|meta|(1583,5)})]),
        cond(equ(var("state"@{|meta|(1515,5)}),
                 call("locked"@{|input|(152,6)},[])),
          call("locked-dispatch"
               {|meta|(1565,9),|input|(152,6)},
               [var("event"@{|meta|(1583,5)})]),
          val(error("UnsupportedState"
            @{|meta|(1375,16)})))))], [])
```

(c) Generated AST of the compiled state machine

Fig. 2: The names in the state machine code (a) end up as strings in the AST (b), the origins of which are propagated to the compiled AST (c). State machine input is represented by URI *input*, the transformation definition by URI *meta*.

syntax tree (AST), which is shown in Figure 2b. Note that all strings in this AST are annotated with their origin, using the pseudo-notation "@". The AST is then translated to an imperative program which is shown in Figure 2c. Some strings have *input* origins (e.g., "opened"), some are introduced by the transformation and have *meta* origins (e.g., "main"), and some strings have origins in both the input and transformation because of concatenation (e.g., "opened-dispatch").

3 Applications of string origins

3.1 Hyperlinking generated artifacts

One of the foremost applications of string origins is relating (sub)strings of the output back to the input of a transformation [12,10]. Applications of this information include embedding links back to the source program in generated code, inspectors, debuggers (e.g., using SourceMaps [15]), or translating back errors produced by further transformations (e.g., general-purpose language compiler errors). In this section we show an example of inspecting the result of a program transformation where the output is shown in an editor with embedded hyperlinks to the input or transformation code.

To display hyperlinks for parts of the generated code, the offsets of the chunks in the generated code must be mapped back to the origin associated with each corresponding

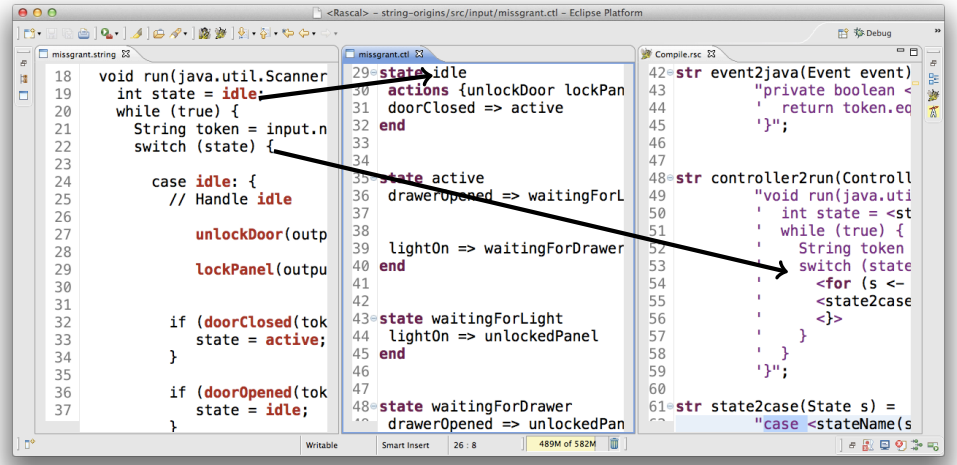


Fig. 3: Three editors showing (1) generated code with embedded hyperlinks (2) the input state machine model and (3) the transformation code. Fragments of the generated code that originate from the input are in bold red.

chunk. Fortunately, the trace relation introduced in Section 2.1 contains exactly this information. The hyperlinks are created by finding the location of a click in the Trace mapping and moving the focus and cursor to the corresponding editor.

A demonstration of this feature is shown in Fig. 3. The screenshot shows three editors in Rascal Eclipse IDE. The first column shows generated Java code. The substrings highlighted in red are the substrings originating from the input, a textual model for state machines (shown in the middle). The other substrings (in black) are introduced by the code generator, which is shown in the right column. Clicking anywhere in the first column will take you to the exact location where the clicked substring originated.

3.2 Protecting regions of generated code

In many cases, a model-to-text transformation is intended to generate just a partial implementation that has to be completed by the programmer. Normally, if the transformation is re-run, the manually edited code is overwritten. In general, this problem is addressed by explicitly marking certain zones of the generated text as *editable*. The MOF Models to Text Standard [11], for instance, introduces the unprotected keyword at the transformation level to specify whether a region can be editable by the end programmer or not. Another traditional solution is the generation gap pattern [6], in which the generated code and the code that is expected to be handwritten are related by inheritance. This, however, demands that the generated code is written in a language that features inheritance and also that the writer of the transformation encodes this design pattern in the transformation.

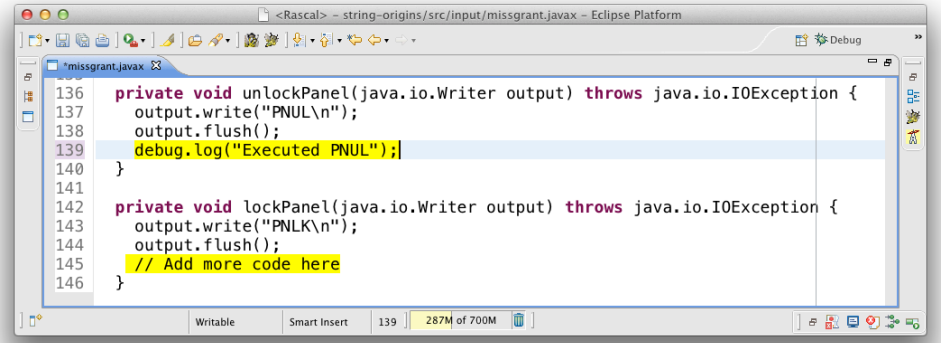


Fig. 4: Editor featuring highlighted editable regions.

String origins allow us to tackle this problem in a language and transformation design agnostic way. Since locations correspond to extended URIs, they can be enriched with meta data in the form of query string parameters. We provide three functions tagString(key,value), getTagValue(key) and isTagged(key), as an abstract interface to these query strings. The tagString function could be used in a transformation to tag regions of text as editable. For instance, the following code snippet marks a substring as being editable in the code generator for a state machine language:

```
str command2java(Command command) =
  "private void <command.name>(Writer output) {
  '   output.write(\"<command.token>\\n\");
  '   <tagString("// Add more code here", "editable", command.name)>
  '}";
```

The function tagString transparently marks the origin of the inserted string (`"// Add more code here"`) to be an `editable` region and names it as the name of the command input to command2java.

To provide editor support for editable regions, the marked substrings need to be extracted from the generated code. The function extract constructs a map from output location to region name using the index function introduced in section 2.1.

```
alias Regions = map[loc pos, str name];
Regions extract(str s, loc l) =
  (l: getTagValue(x, "editable") | <l, x> ← index(s, l), isTagged(x, "editable") );
```

From the index computed on the generated code s and the target location l, the function extract collects all locations which have an associated string value that is tagged as editable. An editor for s can then use the locations in the domain of this map to allow changes to the regions identified by the locations. In fact, it maintains another map, this time from region name (range of the result of extract) to the contents of each region.

When the code is regenerated, the edited contents of the regions need to be plugged back into the newly generated code, to restore the manual modifications. The function plug performs this task:

```
alias Contents = map[str name, str contents];
str plug(str s, loc l, Contents c) = substitute(s, extract(s, l) o c);
```

The Contents type captures the edits made in the editable regions. The function plug uses a generic substitution function (substitute) which receives a map from location to string and performs substitution based on the locations. To obtain this map, plug composes the map returned by extract with the contents c, where the map composition operator o is similar to relational composition.

As a proof of concept, we have added a feature to the Rascal editor framework that uses the presented infrastructure in order to provide consistent editing of generated artifacts with editable areas. When a transformation that produces editable regions is executed, a file with information about the editable offsets is generated as well. When the user opens a generated file, the editor checks if the region information is available. If so, the editor restricts the editing of text just to the regions marked as editable, ensuring that the fixed generated code stays as it is. Fig. 4 shows a screenshot of the editor with highlighted editable regions.

3.3 Resolving symbolic names

Textual DSLs or modeling languages employ symbolic names to encode references, for instance from variables to declarations. As a result, DSL compilers and type checkers require name analysis to resolve references to referenced entities, in fact imposing a graph structure on top of the abstract syntax tree (AST) of the DSL. The names themselves cannot be used as nodes in this graph, since then different occurrences of the same name will be indistinguishable. A solution to this problem is to assign unique labels to each name occurrence in the source code. Since no two names can occupy the same location in the source code, string origins are excellent candidates to play the role of such labels.

Figure 5a shows the abstract syntax of the state machine language used in Fig. 2. Note that states, events and transitions contain strings. Each of these strings will be annotated with an origin by the state machine parser as in Fig. 2b. Figure 5b shows the generic type Ref for *reference graphs*: a tuple consisting of the set of all name occurrences (names), and a relation mapping uses of names to declarations. The function resolve computes a reference graph by first constructing two relations mapping names of states and events to declarations of states and events, respectively (sds resp. eds). The last comprehension uses the deep matching feature of Rascal (/) to find transitions arbitrarily deep in the controller ctl. Each transition then contributes two edges to the relation e.

Reference graphs such as returned by resolve have numerous generic applications in the context of DSL engineering. For instance, reference graphs can be used to implement jump-to-definition hyperlinking of editors: when the user clicks on the use of a name, the reference graph can be used to find the location of its declaration. Another application is rename refactoring: given a reference graph, and the locations of a name occurrence, it is possible to track other names that reference it or are referenced by it and consistently

```
data Controller
  = controller(list[Event] events,
               list[State] states);
data State
  = state(str name,
          list[Transition] trans);

data Event
  = event(str name, str token);

data Transition
  = transition(str event, str state)
```

(a) AST data type of state machines

```
alias Ref = tuple[set[loc] names,
                  rel[loc use, loc def] refs];

Ref resolve(Controller ctl) {
  sds = { <x, origin(x)> | state(x, _) ← ctl.states };
  eds = { <x, origin(x)> | event(x, _) ← ctl.events };
  v = range(sds) + range(eds);
  e = { <origin(e),ed>, <origin(s),sd>
        | /transition(e, s) := ctl,
          <e, ed> ← eds, <s, sd> ← sds};
  return <v, e>;
}
```

(b) Name resolution for state machines

Fig. 5: Implementing name resolution for state machines

rename them. Finally, if Ref is slightly modified to distinguish uses from declarations in the names component, reference graphs can be used to report unbound names or unused declarations.

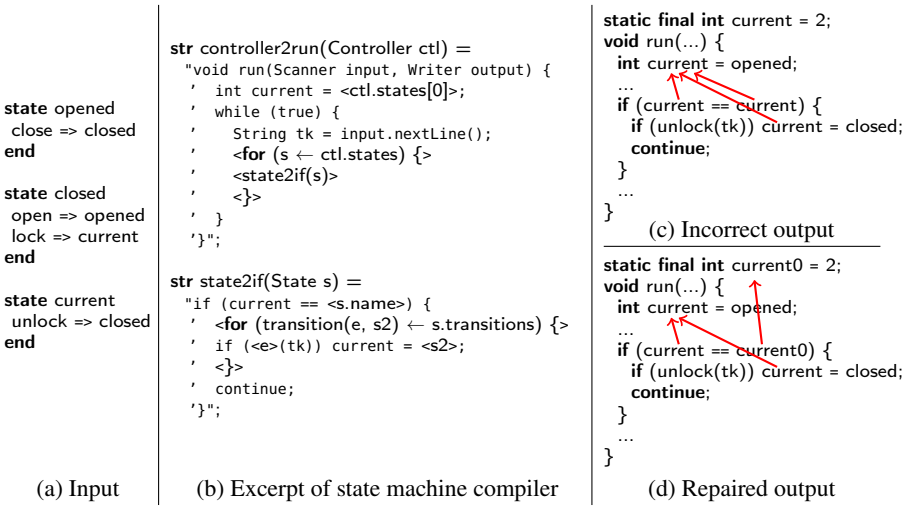
3.4 Enforcing a same origin policy for references

A common problem with code generation is that names used in the input (source names) which pass through a transformation and end up in the output might interact with names introduced by the transformation (introduced names). For instance, the declaration of a name introduced by the transformation might capture a reference to a source name, or vice versa. This is the problem that is traditionally solved in the work on macro hygiene [3].

The problem of inadvertent name capture is best illustrated using an example. Figure 6a shows the simple state machine used earlier in Fig. 2a, but this time the last state is named current. The code generator of state machines – partially shown in Fig. 6b – introduces another instance of the name current to store the current state in the generated Java implementation of the state machine. As a result, the declaration of this current captures the reference to the state constant current.

The reference arrows in Fig. 6c show that both current variables in the if-condition are bound by the current state variable declaration. However, the right-hand side of the equals expression should be bound by the constant declaration corresponding to the state current. Moreover, the Java compiler will not signal an error: even though the code is statically correct, it is still wrong.

To avoid name capture, the algorithm described below renames the source names in the output of a transformation if they are also in the set of non-source names. The result can be seen in Fig. 6d: the source occurrences of current are renamed to current0, and inadvertent capture is avoided. Effectively, the technique amounts to enforcing a same origin policy for names, similar to how a same origin policy avoids cross-site scripting



(a) Input (b) Excerpt of state machine compiler (c) Incorrect output (d) Repaired output

Fig. 6: Example of repairing name capture: the input (a) contains the name current, but this name is introduced in the transformation as well (b). Consequently, the introduced variable in the output shadows the constant declaration (c). The fix function renames all occurrences of current originating in the input to current0 so that capture is avoided (d). The arrows in (c) and (d) link variable uses to their declarations.

attacks in Web application security[1]: names originating from different artifacts should not reference each other.

In [5] the authors showed how string origins proved to be instrumental in automatically repairing the problem of unintended variable capture. In this section we present a technique that is simpler but also more conservative: it might rename more identifiers than is actually needed. Whereas the method of [5] is parameterized in the scoping rules of both source and target language, the technique of this section is language agnostic, and does not require name analysis of the source or target language.

The key observation is that whenever name capture occurs it involves a source name and a name introduced by the transformation. This difference is reflected in the origins of the name occurrences in the output: the origins' source locations will have different URIs. The same origin policy then requires that for every reference in the generated code from x to y, both x and y originate from the input or neither. The same origin policy is enforced by ensuring that the set of source names is disjoint from the set of names introduced by the transformation. This can be realized by consistently renaming source names in the generated code when they collide with non-source names.

To formalize the same origin policy, let $t = f(s)$ be the result of some transformation f on input program s, inducing a trace relation $\tau \in \mathit{Trace}$, and let $G_s = \langle V_s, E_s \rangle$, $G_t = \langle V_t, E_t \rangle$ be the reference graphs of the source s and target t, respectively. The same origin

[1] http://en.wikipedia.org/wiki/Same-origin_policy

```
str fix(str gen, Index names, loc inp) {
  bool isSrc(str x) = origin(x).path == inp.path;
  set[str] other = { x | <_, x> ← names, !isSrc(x) };
  set[str] allNames = { x | <_, x> ← names };
  map[loc,str] subst = ();
  map[str,str] renaming = ();
  for (<l, x> ← names, isSrc(x), x in other) {
    if (x notin renaming) {
      <y, allNames> = fresh(x, allNames);
      renaming[x] = y;
    }
    subst[l] = renaming[x];
  }
  return substitute(gen, subst);
}
```

Fig. 7: Restoring disjointness by fixing source names.

policy then requires that

$$\forall \langle l_1, l_2 \rangle \in E_t, \langle l_1, o_1 \rangle \in \tau, \langle l_2, o_2 \rangle \in \tau \,:\, o_1 \in V_s \Leftrightarrow o_2 \in V_s$$

To enforce the same origin policy, one more assumption on reference graphs is needed, namely that the locations in every reference edge point to the same textual name. In other words: every use is bound by a declaration with the same name. For instance, the reference edges drawn in Fig. 6c and Fig. 6d satisfy this invariant since variable uses l_1, l_2, l_3 point to occurrences of the name current, which is also the name used in the declaration l_0.

If we assume that the same name invariant is true for E_t, then the same origin policy is satisfied if the set of source names is disjoint from the set of names introduced by the transformation. The same name invariant ensures that for every $\langle l_1, l_2 \rangle \in E_t$, we have that l_1 and l_2 point to the same name. Consequently, it is not possible that one name originates from the input (e.g., through o_1) but the other does not (e.g., through o_2) because that would contradict disjointness of names.

The code for restoring disjointness is shown in Fig. 7. The function fix has three parameters: the generated code gen, the index names capturing the names occurring in gen, and a source location identifying the input program inp. The latter is used by the predicate isSrc to determine whether a name x is a source name by checking if the path in the origin of x is the input path.

The **for**-loop iterates over the index names that represents all names in the generated string gen. If such a name x originates in the source and is also used as an other name, an entry is created in the substitution subst, mapping location l to a new name. The new name is retrieved from the renaming map which records which source names should be renamed to which new name. The function fresh produces a name that is not used anywhere (i.e., it is not in allNames). The variable allNames is updated by fresh to ensure that consecutive renames do not introduce new problems.

Note that fix could also be parameterized with an additional set of external names which might capture or be captured by source names. External names could include the reserved identifiers (keywords) of the target language or (global) names that are always in scope (e.g., everything in java.lang). The only required change is to add the external names to other.

4 Implementation

The implementation of string origins requires changes to the internal representation of strings used by the transformation engine. In this section we discuss the implementation of string origins in Rascal.

As Rascal is implemented in Java, we have implemented string origins in Java as well. Rascal string values (of type **str**) are internally represented as objects conforming to the interface IString. We have reimplemented this interface to support string origins, changing only the internal representation. Instances of IString are constructed through a factory method IString string(java.lang.String) in the Rascal factory interface for creating values (IValueFactory).

To ensure that the propagation of string origins is complete, every created string now needs a location to capture its origin. We have extended IValueFactory with another factory method IString string(java.lang.String, ISourceLocation) to support this. Calls to the original string(...) method were changed to the new one, everywhere in the Rascal implementation. The locations where changes have been made correspond to the following three categories:

- Input: any function that reads a resource into a string must be modified to install origins on the result. In Rascal, these are built-in library functions like readFile(**loc**), readLines(**loc**), parse(**loc**), etc.
- String literals: constant string values that are created as part of a Rascal program get the origin of the string literal in the Rascal file. Whenever a string literal is evaluated, its location is looked up in its AST and passed to the factory method. This category also covers interpolated string templates.
- Conversions: converting a value to a string in Rascal is achieved through string interpolation. For instance, "<x>" returns the string representation of x. If x evaluates to a string, the result of the conversion is that string itself (including origin); otherwise, the newly created string gets the locations of the expression x in the Rascal source.

String origins are propagated through all string operations. As a result, all operations provided in the IString interface have been reimplemented. The two most important operations are concat and substring. Their semantics is illustrated in Fig. 8. The top two string values are annotated with source locations |file:///foo.txt|(0,5) and |file:///bar.txt|(0,5). Concatenating both strings (middle row) produces a new, composite string, where the original arguments to concat are still distinguishable, and have their respective origins. Finally, the substring operation computes a new composite string with the origin of each component updated to reflect which part of the original input is covered. Besides concat and substring, all common string operations such as indexOf, replaceAll, split etc. can be defined on strings with origins, with full propagation.

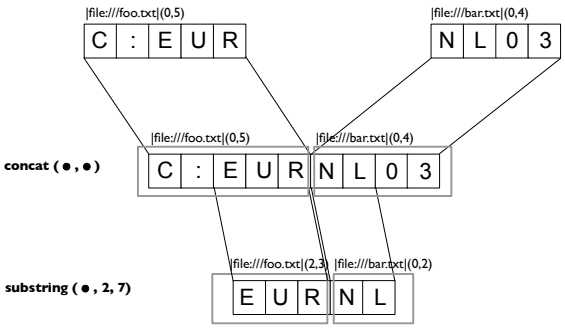


Fig. 8: The concat and substring operations defined on origin strings

Internally, Rascal strings with origins are represented as binary trees. A string is either a *chunk* object which has a source location attached to it, or it is a *concat* object which represents two concatenated strings. A string represented as a binary tree can be flattened to a list containing elements with a string value and source location for each chunk object at the leaves. This list is the basis for the functions index and origin introduced in Section 2.1.

Although in our experience the performance penalty introduced by representing strings as binary trees is acceptable in practice, further benchmarking is needed to assess the overall impact. In particular, it will be interesting to see how the choice of representation affects different use cases. For instance, when generating code, concatenation is one of the most frequently executed string operations. The binary tree representation is optimized for that: concatenation is an $O(1)$ operation. On the other hand, analyzing strings (e.g., substring, parsing, matching) is much more expensive if a string is a binary tree. But then again, the penalty will be most significant if these operations apply to strings resulting from concatenation in the first place. We consider investigating these and other aspects of performance an important area for further research.

5 Related work

String origins are related to previous work in origin tracking, taint propagation and model traceability in model-driven engineering. Below we discuss each of these areas in turn.

Origin tracking. The main inspiration of string origins is origin tracking [16]. In the context of term-rewriting systems, this technique relates intermediate subterms matched and constructed during a rewriting process. Origin tracking was proposed as a technique to construct automatic error reporting, generic debuggers and visualization in program transformation scenarios. String origins are related in that the result is a relation between input and output terms. However, for string origins, only string valued elements are in this relation. Furthermore, the origin relation of [16] is derived from analyzing rewrite rules. As a result the transformation writer is restricted to this paradigm. With string origins, a transformation can be arbitrary code.

Taint propagation. In Web applications, untrusted user input might potentially end up as part of a database query, a command-line script execution or web page. Malicious input could thus compromise the system security in the form of code injection attacks. Taint propagation [7] is a mechanism to raise the level of security in such systems by tracking potentially risky strings at runtime. It consists of three main phases: mark certain *sources* of strings as tainted, propagating taint markers across the execution of the program, and disallowing the use of tainted strings at certain specific points called *sinks*. The propagation is achieved by annotating the string values themselves and making sure that string operations propagate taintedness.

Although in general the taint information is coarse-grained: any string that is computed from any number of tainted strings is tainted as well. A finer granularity is employed in character-based taint propagation [2]. String origins are very similar to this approach in that the origin is known for each character in a string. On the other hand, string origins can be considered more general, because origins capture more information than just taintedness. In fact, taint propagation could easily be realized using string origins by considering certain input locations as tainted.

In [4], the authors present an application of taint propagation to the domain of model-to-text transformations, specifically, to support debugging of failures introduced in a transformation. Their approach consists in instrumenting the transformation in order to add so-called *tainted marks* to each identifiable element of the input. On the other hand, the user of the transformation has to identify erroneous sections in the output. Since the taints from the input are consistently propagated by the instrumented transformation, it is possible to relate the errors in the output to specific elements of the input. In this work, the input is an XML document and the transformation, an XSLT file. The granularity of this technique is at the level of XML nodes, which provides quite precise information for the error tracking analysis.

Traceability in model-driven engineering. In model-driven engineering, models are refined through transformations to produce executable artifacts. In [1], the authors argue for the need for automatic generation of trace information in such a setting. Several endeavors towards this goal have been reported in the context of different model transformation systems, such as ATL, MOF, and Epsilon.

For instance, ATL transformations can be manually enriched with traceability rules that conform to a traceability metamodel [8]. Besides the target models, the enriched transformations will also automatically produce trace models when executed. In order to avoid the manual work of adding these specifications to existing transformations, the authors present a technique for automatically weaving the trace rules into the transformation. Unlike string origins, this approach relies on the structure of the ATL rules to derive the trace links, and such links just relate a subset of the elements in the target model to certain elements in the source model, but not to the transformation itself.

Another approach to address traceability is the MOF Models to Text Transformation Language standard [11]. In this specification, transformations can be decorated with a `trace` annotation so when the transformation is executed, a relation between its output and its input is constructed. As in the case of [8], the transformation conveys the traceability information explicitly. To overcome this, [12] and [13] introduce an alternative technique for managing traceability in MOFScript, a language for defining model to text transfor-

mations based on the MOF standard. In this case, "any reference to a model element that is used to produce text output results in a trace between that element and the target file". Like string origins, this technique provides implicit propagation and fine-grained tracing. However, no relation between the output and the text fragments coming from the transformation is created. Just as in the case of ATL, MOFScript depends on the structure of the rules to analyze the transformation and generate trace information.

Finally, The Epsilon Generation Language (EGL) is a model-to-text transformation language defined at the core of the Epsilon Platform [14]. EGL provides an API to construct a transformation trace. However, this API is coarse-grained (file-level).

6 Conclusion

String origins identify the exact origin of a fragment of text. By annotating string values with their origins, the origins are automatically propagated through program transformations, independent of transformation style or paradigm. The result is that for every string valued element in the output of a transformation, we know where it came from, originating in the input program or introduced by the transformation itself.

String origins have diverse applications. They address traditional model traceability concerns by linking output elements to where they were introduced. We have shown two applications in this space, namely hyperlinked editors for generated code and protected regions. Moreover, string origins can be used to uniquely identify sub terms, which is instrumental for implementing name resolution, rename refactoring, jump-to-definition services and error marking. Finally, we have shown that by distinguishing source names from introduced names, accidental name capture in generated code can be avoided in a reliable and language agnostic way.

The implementation of string origins is simple and independent of any specific meta-model, transformation engine or technological space. Any transformation system or programming language that manipulates string values during execution can support string origins by changing the internal representation of strings. The standard programming interface on strings remains the same. As a result, code that manipulates strings does not have to be changed, except for the code that creates strings in the first place. Although conceptually simple, we have shown that string origins, nevertheless, provide a powerful tool to improve the understandability and reliability of program transformations.

References

1. N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Syst. J.*, 45(3):515–526, July 2006.
2. E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *Proceedings of the 2009 ACM workshop on Secure web services*, pages 3–12. ACM, 2009.
3. W. Clinger and J. Rees. Macros that work. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 155–162. ACM, 1991.
4. P. Dhoolia, S. Mani, V. Sinha, and S. Sinha. Debugging model-transformation failures using dynamic tainting. In T. D'Hondt, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 26–51. Springer Berlin Heidelberg, 2010.

5. S. Erdweg, T. van der Storm, and Y. Dai. Capture-avoiding and hygienic program transformations. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2014. To appear.
6. M. Fowler. *Domain-Specific Languages*. Addison Wesley, 2010.
7. V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Computer Security Applications Conference, 21st Annual*, pages 9–pp. IEEE, 2005.
8. F. Jouault. Loosely coupled traceability for atl. In *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29–37, 2005.
9. P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain-specific language for source code analysis and manipulation. In *Proceedings of Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177, 2009.
10. D. S. Kolovos, L. Rose, R. Paige, and A. García-Domínguez. The Epsilon book, 2012. Available at `http://www.eclipse.org/epsilon/doc/book/`, accessed Nov. 13, 2012.
11. Object Management Group (OMG). MOF Model to Text Transformation Language 1.0. formal/2008-01-16, January 2008.
12. J. Oldevik and T. Neple. Traceability in model to text transformations. In *2nd ECMDA Traceability Workshop (ECMDA-TW)*, pages 17–26, 2006.
13. G. Olsen and J. Oldevik. Scenarios of traceability in model to text transformations. In D. Akehurst, R. Vogel, and R. Paige, editors, *Model Driven Architecture- Foundations and Applications*, volume 4530 of *Lecture Notes in Computer Science*, pages 144–156. Springer Berlin Heidelberg, 2007.
14. L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack. The Epsilon generation language. In *Model Driven Architecture–Foundations and Applications*, pages 1–16. Springer, 2008.
15. R. Seddon. Introduction to JavaScript source maps. Online, 2012. `http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/`.
16. A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Symbolic Computation*, 15:523–545, 1993.