s5varma Work Log

Contents

s5varma Work Log

- Author: Sunjay Varma
- Email: s5varma@edu.uwaterloo.ca

- Term: January 2019 - April 2019

Apr 24, 2019

Merge Request: https://git.uwaterloo.ca/swag/Rex/merge_requests/12

Separate Compilation and Linking

Given that Rex is designed to analyse each source file without making any assumptions about other source files, it makes sense that you should be able to analyse each file individually without having to run everything together. We used this assumption when we parallelized the execution of Rex a few months ago. This is incredibly useful for both users of Rex and its developers because that means that they can potentially run Rex on just a few files instead of on an entire project over and over again.

For users, if only a single file or a few files have changed, they can re-run Rex on just those files. For developers of Rex, if a bug occurs when analysing a single source file but you need to check the result in the final TA file, you should be able to do that without re-analysing everything.

The goal of the separate compilation and linking work is to enable exactly this functionality.

Implementing this feature completely changed the architecture of Rex from being a primarily interactive, REPL-based tool to a fairly standard command line application similar to many compilers. The `FactExtractorManual.pdf` found in the `docs/` folder of the Rex repository covers the installation and usage of Rex as it is right now.

Rex analyses files in two stages:

1. **Separate compilation**

   Each source file is analysed using our walkers and the resulting TA graph is written to a TA Object File (extension `.tao`) specific to that source file. The `.tao` file format is a custom, Rex-specific format designed to make the Linking step very fast. The key thing to remember about `.tao` files is that they contain all the information of a TA file: nodes, edges, and node/edge attributes. The schema information is assumed and thus omitted from the `.tao` file.

2. **Linking**

   In this stage, each `.tao` file passed implicitly/explicitly to Rex is processed to produce a final TA file. The majority of the rest of this entry will dive into the details of the linking process and how the `.tao` files are designed to enable it.

Linking: TA File

Background

Most traditional linkers operate on binary files called Object Files. These files typically have the extension `.o` and can be produced by all useful C/C++ compilers upon request. These files contain what is known as

Object Code. Object Code is an incomplete form of the full code of the executable. The linker's job is to take that form and use the information from other object files to build a complete executable that can be run on a given target machine.

The reason I'm describing this to you (despite you probably already knowing all of this) is because Rex's linking process is intentionally designed to largely mimic this same process as common C/C++ compilers/linkers.

The difference is that instead of source code, we're working with TA graphs. TA graphs are directed graphs with colored edges, not source files, so it isn't immediately obvious that you would be able to link them in the same way you link source code. There is no obvious way to link most general directed graphs that you might see in a math course.

The key observation we've made through our work is that you can do this for our TA graphs because the nodes we are working with directly correspond to various parts of source files. That means that there is an obvious mapping between nodes in different graphs. If a node from one graph corresponds to the same source location as another graph, we can link (re: merge) those two nodes together.

Since our problem is related to the problems already solved by modern compilers, we can leverage their work in our implementation. Most linkers follow a simple, two-pass process:

1. **First pass:** go through each object file and create a symbol table
2. **Second pass:** go through each object file again and use the symbol table to perform linking as you generate code for each object file

The generated code from the second step is written to an executable that can then be run on its target machine.

While the process itself can be described in two steps, the actual content of each step is incredibly complex. Linkers need to worry about all kinds of issues with both symbol resolution and relocation (among other things).

Luckily for us, we will have far simpler problems because we can control each step of the process (including the object files) in order to make this extremely efficient.

Goals

Before we get into that though, let's consider *why* the linking process was probably designed this way. Consider that linkers need to work even if the codebase they are linking is incredibly large. The generated object code for a huge project may be much larger than the available RAM on a user's computer. This is both possible today and was incredibly common back when linkers were first invented. A consequence of this size issue is that a linker must be designed to perform its operations **without loading all of the object files into memory.**

This problem is present for us in Rex as well. Before this new architecture was implemented, we would keep the generated facts for **all** files in a user's project in memory. To be fair, the size of our TA factbases is *much* smaller than the total size of the object files for a user's project. While linking will save on some of our memory usage, a much greater gain for us is the ability to re-run partial analyses and link the results as if we had run the entire thing.

With the C/C++ linkers in mind, the Rex linking process was designed to link our generated facts while minimizing memory usage as much as possible. We avoid loading the generated facts in memory and instead keep everything on disk in special "TA Object Files". The TA Object Files have the extension `.tao` and are written in a Rex-specific format that is specially designed to make our linking process as efficient as possible.

A nice consequence of this design is that we can more easily scale up the number of facts that we generate without drastically increasing the overall memory usage of Rex. Adding facts will only ever increase the size of the `.tao` file for a given source file and will not affect the memory usage of Rex during the linking process.

> **Terminology Note:** An **instance** is a node in a TA graph. These are typically declared with an `$INSTANCE` edge in a TA file. A **fact** is an "edge" in a TA graph. There are times in my writing where I will be sloppy and use the word "facts" to refer to both "instances and facts". This is usually just in places where we are considering the scale of things in a TA file. The number of facts is usually far greater than the number of instances, so it is easy to group the two when describing a runtime.

Assumptions

In our design and analysis of this linking algorithm, we've made a number of assumptions that are important to be aware of. Note that this list may be incomplete. I have tried to cover the most critical ones, but may have missed some assumptions that were either made implicitly or never written down.

1. We are assuming that the number of instances and facts generated in any one file is far less than the total number of instances and facts in the project. That means that walking each file while retaining all nodes/edges for that file in memory is fine. If any one file is way too big, this assumption is broken.

2. We are assuming that the file names passed to Rex are unique. Any overlap will cause a data-race while writing the .tao file. This assumption only affects the implementation of linking, but is still important to know about. Overlap is pretty easy to encounter. If a project has two directories that contain a source file with the same name (e.g. `main.cpp`), the generated `.tao` file will have the same name for both. This overlap is dangerous because it can result in facts being lost or not included because the `.tao` file for one of the source files was overwritten.

3. We are assuming that there are no duplicate instances or facts extracted from compilation units. If this is the case we may end up with duplicate nodes or edges. The potential for this problem existed in Rex before we implemented separate compilation/linking. In the future we may extend Rex to intelligently handle such situations.

4. Similarly, we assume that no single `.tao` file has multiple declarations of the attributes for a given node/edge. We need this because otherwise the process of linking attributes would need to be much more complicated. We probably would not be able to do it as fast if this assumption were broken.

   In this case, this assumption is actually always true by design. In our implementation of attributes in Rex, we keep the attributes in a hash map while walking the AST. Duplicates are not possible because they would hash to the same value as each other. That being said, there is a related problem that is a consequence of this. Attributes can be overwritten during analysis and that can cause us to lose information that we have otherwise recorded. A task has been added for that and we will likely address this in the future.

5. This particular assumption defines a number of useful values for asymptotic runtime and memory usage analysis. The total number of nodes generated from all source files is `V`. The total number of edges generated from all source files is `E`. Given that a node can and often will be part of multiple edges, we can assume that `E` is far greater than `V`. The total number of attributes for a given node/edge is bound by some constant `K`. `K` is much less than the number of nodes/edges `M` in a given source file that have any attributes at all. `K` is also less than the number of source files `N`. The number of source files is the same as the number `.tao` files passed to the linker. The number of source files `N` is far less than either `E` or `V`.

   `K` being a small constant is useful because it allows us to eliminate it in most analyses involving any of the other factors. The reason we can do this is because `K` is usually quite small. In practice, most nodes/edges that I have observed have one or two attributes at most. I would wager that all nodes/edges generated by Rex have less than 5-6 attributes. Even if this turns out to be incorrect one day, `K` would have to grow to the size of `N` or `M` in order for its value to really matter.

With these assumptions written out and given the variables defined in assumption 5, we can more precisely define the characteristics of the linking algorithm designed as part of this work:

> The original implementation of Rex used `O(V + E)` memory. All nodes and all edges were kept in memory at once. This is the same as `O(E)` memory since `E >> V`.
>
> In contrast, the linking process described below uses `O(V)` memory. The only memory kept around for the entire duration of the process is a symbol table containing `O(V)` nodes. Edges are loaded, but more than one is never kept in memory at any given time. There are parts of linking that take `O(N)` extra memory, but given that `V >> N`, we still use `O(V)` memory overall.

This is approximately proportionate to the memory usage you would expect a C/C++ linker to have given its two pass algorithm. In the next section, we'll see how this was accomplished.

The Linking Algorithm

Finally, after so much build up, we can get to the actual linking part of this log entry. The overall description of the algorithm is actually not that interesting because it is almost exactly the same as the two-pass process described for C/C++ linkers. This is intentional. We want to leverage the innovations of our predecessors, not design something arbitrarily from scratch and re-make all the same mistakes they already have.

The interesting part is in the details of how the linking passes are implemented. The implementation of this process is heavily optimized to the point where we actually only make a total of one pass over all the TA Object Files. We still make two passes over the list of files we need to go through, but the files are kept open throughout and we only end up reading through them all once.

The two passes in our linking algorithm are as follows:

1. First pass: go through the `.tao` files and create a symbol table that can be used to look up nodes
2. Second pass: go through the edges, node attributes, and edge attributes and only write out the ones that exist in the symbol table

If you read the implementation of these passes, you'll see that the second pass is actually expanded into multiple small passes. We need to do this because the TA file has discrete sections. If we were linking into a graph database, the algorithm could truly be done in exactly two passes.

Note: The description of the algorithm in this log entry is fairly high level. To understand all the details, see the reasonably well commented implementation of the linker in the `Linker/` directory of the Rex source code.

In pseudo-code, the linking algorithm is as follows:

```
Input:
- N open file streams for each .tao file
- The output file stream for the generated TA file

First pass:
- Let S[id] be a set of node IDs that have been "established"
- S is initially empty and has constant time insertion and lookup
- For each .tao file input stream:
  - Read each node in the .tao file one by one and insert them into S[id]
  - As each node is read, output it to the TA file with an $INSTANCE line
    (i.e. "create" the node)
- S[id] now has size O(V)

Second pass:
- For .tao file input stream:
  - Read a each edge one by one
  - Establish the edge: Check if both node IDs in the edge are in S[id]
  - If so, write it to the output TA file (i.e. "create" the edge)
```

This process is (intentionally) very simple. In fact, the entire implementation of this linking algorithm (not including comments) is only about ~200 lines of code. The benefit of leveraging the work of the linker writers of the past is that while a great deal of thought went into this process, the result is relatively easy to follow.

Overall, as you should be able to determine from the pseudo-code above, this algorithm uses `O(V)` memory exactly as promised. Only a single object of exactly that size is built up throughout the entire process.

There is no clear way to reduce that drastically without costing us enormous amounts of runtime. For example, we could store the symbol table on disk (maybe in partitions), but we would only ever save a linear factor less than `V` and never do better asymptotically. All of that said, the process above is clearly much better than the `O(V + E)` memory usage we had before given that `E >> V`.

Finally, now that we have covered the base algorithm, we should discuss its details. There is lots behind the simplicity of this algorithm that hasn't been covered yet. For example, we used the term "establish", but didn't describe what that means formally. This algorithm also heavily relies on the concept of a unique node ID, so it's worth discussing how we accomplished that. The node ID is actually the entire backbone of this process and ends up radically simplifying a lot of aspects of linking that C/C++ linkers have to do much more about. The linking algorithm above also didn't cover linking node/edge attributes. That subprocess is very interesting and highly optimized as well. It doesn't increase the asymptotic memory usage of this algorithm or change its core implementation, so I've left it to a later section.

Establishing Nodes and Edges

A key distinction in most linkers is between a **strong symbol** and a **weak symbol**. A strong symbol must be resolved during the linking process, whereas a weak symbol is not required to be. For Rex, we treat **all** symbols as weak symbols. This is necessary because our symbol type is the node ID and we generate way more nodes than we actually want to keep.

Any normal-sized C++ program will use the standard library and maybe even other external libraries. We may be analysing all of its code or just a small subset. The reason we choose to only have weak symbols is because the edges we generate very often end up containing node IDs from these libraries or from files we are not analysing. All of that needs to be filtered out so we only end up with facts core to the analysis we are trying to perform.

A consequence of this is that if we implement things incorrectly or misconfigure a run of Rex, we can sometimes inadvertently filter out facts we actually would have wanted to keep.

The term **established** refers to an **edge** where both nodes are from the set of files that we want to keep. Sometimes I may refer to a node as being established. This is just a short way of saying that the node comes from a file that is in the set of files that we want to keep. If a node is established, it can be used to establish an edge.

The question is, how can we determine which nodes or edges to establish while we're walking the files? Rex will analyse anything you provide to it without really considering which file it is in. The Clang API walks headers and source files without considering whether they are part of the file we originally intended to analyse. (It is actually good that Clang does this or we would have no information in the Clang AST.)

The approach we've taken to determining whether a node should be kept is very straightforward: When we create a node, we pass in a flag (a boolean) called `shouldKeep`. If `shouldKeep` is true, we keep that node. We determine the value of this boolean by asking Clang if the boolean is in the "main file". The "main file" is a concept in the Clang API that represents the file used as the basis for the current compilation unit. The current compilation unit can be composed of source files and header files–anything that was included. The main file is the file we passed into Clang to analyse in the first place.

The `shouldKeep` flag is used by the code that outputs the TA Object File. We only output nodes that have this flag set to true. That means that when the linking process above is run, only nodes that we have established are added to the symbol table. That means that when we look up nodes to establish an edge, only those nodes will be able to do that.

The main file check is not perfect. For example, there are situations in C++ where while a declaration may exist in that file, semantically it shouldn't be considered as actually being "in" that file. Both functions without bodies (i.e. forward declarations) and variables with external linkage are not considered "in" the main file even if there declarations are there. This is because even in a C/C++ linker, those declarations would need either an implementation or an initialization somewhere else.

Classes are a complicated issue because in an ideal world we would only include classes that are actually implemented in files that we are analysing. Without being able to determine that, we could potentially include class nodes for classes that are only in header files but not actually implemented anywhere. Thanks to the main file check (as well as some other unmentioned filtering), we are still safe from accidentally including system file classes.

Another complication introduced by this concept of "established" edges and nodes is that we need to include header files as part of our analysis. It used to be sufficient to just pass in CPP files. Without header files, the main file check would never return true for the majority of class/method declarations since those typically all take place in headers. We need to analyse the header files specifically so we have a chance to treat them as a main file and collect all of their declarations.

Structured Unique IDs

In order for the linking process to work, it needs to be able to reliably distinguish between nodes that are the "same" and nodes that are not. This concept of "sameness" is related to "uniqueness", however just having one or the other is not very useful. For example, we could have generated node IDs using a per-source-file counter. Each node would get a successive integer ID and that would be how we distinguish them. The problem with this is that while the node IDs would be unique, we would have to store a lot of additional metadata in order to determine if two node IDs represent the same node. This is not a very efficient way to implement this. It would greatly complicate the linking process as well.

Thus, to determine whether two IDs represent the same node, we developed an "ID scheme". This "ID scheme" is just a well thought out way of generating IDs for Rex that encapsulates the "structure" of the item represented by the node.

By doing this, we can compare just the node IDs of two nodes and determine whether they represent the same item in C++. A neat product of this work is that by carefully choosing how we generate IDs, we managed to handle external linkage (`extern int x;`), internal linkage (`static int x = 2;`), and orphaned forward declarations (`int foo(int x);` in the CPP file, not a header) without any additional complexity in the linker. The linker only knows about nodes and edges, and is completely unaware of anything related to C++ or ROS.

This is a huge accomplishment because we originally expected to need to build a more sophisticated symbol table based on knowledge of C++ declarations. We were not expecting that a sufficiently clever ID scheme would enable us to do without that.

It is certinaly possible that a future version of the linker will need to be more sophisticated. The point of writing all of this isn't to discourage that. It's just very important to notice how much the representation of your symbols can simplify the name resolution process. Keep that in mind when modifying the ID scheme to support future features.

When we were developing the ID scheme, it was determined that Rex nodes store two different "kinds" of items:

- C++ class/function/variable/parameter/etc. declarations
- ROS topics

Note the use of the word "kind" instead of "type". There are many more node "types", but each of them only stores one of each "kind" of item.

If you consider the main purpose and functionality of Rex, this makes sense. We extract facts about C++ programs. All C++ entities are in some way tied to a declaration. We also extract facts about the ROS message passing facilities. In order to do that we have to capture the notion of a "topic" which is distinct from any declaration in C++.

ID Scheme for C++ Declarations A key concern to keep in mind when designing ID schemes is that Rex supports analysing multiple executables together. This is necessary for us to be able to support the analysis of message/data flow through multiple components of a software landscape. That means that the usual assumption that global names must be unique in C++ can be violated. Global names must be unique within a particular executable, but two executables are allowed to have global items with the same names. For Rex's analysis to be accurate, it must distinguish between the global names of different executables.

We also need to keep in mind the following aspects of C++:

- Names within different functions can be the same
- Classes can have methods with the same name that are still distinct because the class they are part of is different
- Classes & methods can be declared within functions (not just globally)

A natural way to deal with all of these is to consider the "namespace" or static scope of an item when generating its ID. That is, we can walk up the AST and record everything above a given item to generate its fully qualified name.

A final concern to us when generating IDs is internal linkage (via the `static` keyword). With static linkage, an item is only available in the compilation unit it is declared in. Since a compilation unit can be uniquely identified with its path, a simple way to deal with these is to add that path to the ID. That way multiple static items in different compilation units will have unique IDs.

The formal definition of the ID scheme for C++ declarations is as follows:

```
An ID for a C++ declaration consists of the following 3 parts:

1. The "feature name" (usually the name of the executable)
2. The fully qualified name (accounting for classes declared in functions and
   function overloading semantics)
3. (Optional) The filename of the compilation unit for items with internal
   linkage
```

A subtle part about this is that in order for it to be correct, the filename (part #3), must be considered part of the fully qualified name when generating the value for part #2 of the ID. Without this, variables *within* `static` functions would all be considered the same even though they belong to distinct items. This also means that part #2 is iterative/recursive. We go up the tree and generate the fully qualified name of the parent in order to generate the fully qualified name of the current item.

Let's see some concrete examples of this scheme in action to see that this works. Each declaration in the code sample below is annotated with the case it represents.

```
// featA, foo.h

class Foo {              // case: class (external linkage)
public:
  void foo(int x);       // case: member function (no body, forward declaration)
};

// featA, foo.cpp

#include "foo.h"
```

```
extern int global;        // case: extern variable

void fwdDecl(int y) {     // case: implementation of forward decl from `main.cpp`
  int b = y / 4;          // case: local variable with same name elsewhere
}

static int local = 3;     // case: static variable (internal linkage)

static int spam(int q) { // case: static function (internal linkage)
  int b = q + 5;          // case: local variable with same name elsewhere
  return b;               //       (internal linkage)
}

void Foo::foo(int x) {    // case: member function (external linkage)
  int b = x * 2;          // case: local variable with same name elsewhere
}

// featA, main.cpp

#include "foo.h"

void fwdDecl(int y);      // case: orphaned forward declaration

int global = 2;           // case: global variable (external linkage)

static int local = 1;     // case: static variable (internal linkage)

int main() {              // case: the entry point (present in each executable)
  int b = 21;             // case: local variable with same name elsewhere

  Foo f1;                 // case: local variable in `main`
  f1.foo(b);

  fwdDecl(b);
}
```

The case where a class is declared within a function is not shown here in the interest of brevity. Consider that an exercise left to the reader.

Now let's see the node ID that gets generated for each declaration:

```
// featA, foo.h

class Foo {               // "decl;featA;Foo"
public:
  void foo(int x);        // "decl;featA;Foo::foo(int x)"
                          // "decl;featA;Foo::foo(int x)::x" (for the parameter)
};

// featA, foo.cpp

#include "foo.h"

extern int global;        // "decl;featA;global"
```

```
void fwdDecl(int y) {     // "decl;featA;fwdDecl(int y)"
                          // "decl;featA;fwdDecl(int y)::y" (for the parameter)
  int b = y / 4;          // "decl;featA;fwdDecl(int y)::b"
}

static int local = 3;     // "decl;featA;local;static;foo.cpp"

static int spam(int q) { // "decl;featA;spam(int q);static;foo.cpp"
                          // "decl;featA;spam(int q);static;foo.cpp::q" (for the parameter)
  int b = q + 5;          // "decl;featA;spam(int q);static;foo.cpp::b"
  return b;
}

void Foo::foo(int x) {    // "decl;featA;Foo::foo(int x)"
                          // "decl;featA;Foo::foo(int x)::x" (for the parameter)
  int b = x * 2;          // "decl;featA;Foo::foo(int x)::b"
}

// featA, main.cpp

#include "foo.h"

void fwdDecl(int y);      // "decl;featA;fwdDecl(int y)"
                          // "decl;featA;fwdDecl(int y)::y" (for the parameter)

int global = 2;           // "decl;featA;global"

static int local = 1;     // "decl;featA;local;static;main.cpp"

int main() {              // "decl;featA;main()"
  int b = 21;             // "decl;featA;main()::b"

  Foo f1;                 // "decl;featA;main()::f1"
  f1.foo(b);

  fwdDecl(b);
}
```

> **Implementation Note:** This detail is omitted here, but each ID generated by Rex ends with the symbol `;;` to make parsing the ID easier during linking. By the time I realized that we wouldn't need to parse the IDs during linking, all the code to do that had already been written. If it happens that we never need to do that, this code can all be deleted.

The first thing to notice about all of these IDs is that they are prefixed with `decl;`. The reason for this is because this allows us to dispatch on the "kind" of ID. The IDs for topics are prefixed with `topic;`.

The next thing to note about that all of these IDs is that, by construction, they cannot possibly conflict with the IDs coming from a different feature name or executable. By prefixing each ID with the feature name, we create a unique namespace for each feature. This allows Rex to analyse multiple executables and link the results with no conflicts.

Classes, functions, and global variables all have external linkage by default. That means that they must be unique to a given feature name/executable. By explicitly choosing to *not* include their filename in the fully qualified name, we support this by default as well. There is no need to specially handle forward declarations versus implementations in the ID generation code because they both just end up being linked together due to them having the same ID. It doesn't even matter if a forward declaration is "orphaned" or not. That is,

the forward declaration can be completely isolated from its implementation and all of this will still work.

Handling `extern` variables is the same thing. No external linkage-specific code because the IDs are sufficient for linking the declarations together.

Declarations that have internal linkage (via `static`) do require some special handling. By putting the filename of the compilation unit into the ID of the static function/variable **and** all of its descendents, we make all of those nodes specific to that particular compilation unit.

Since functions can be overloaded in C++, we include the parameter types and declarations in the ID for the function and its descendents.

Notice that despite the fact that the symbol `b` was used as a local variable in every function body in this example, its IDs still captures that each declaration is specific to the static scope that it is declared in.

Every executable has an entry point called `main`. We don't have to handle that specially at all because we prefix each ID with the feature name/executable. The main function is processed just like any other function with external linkage.

ID Scheme for ROS Topics Generating IDs for topics is relatively straightforward. Although Rex supports multiple exectuables, the messages sent between them share topic names. If you send a message to a channel with topic "sensor_data", you expect that subscribing to that topic in another executable using the same "sensor_data" topic name will result in your receiving that message. Given that we want to model these exact semantics, we do not scope the topic name to the executable like we do with C++ declarations.

Thus, the ID scheme for a ROS topic is very simple:

`topic;<topic name>`

The `topic;` prefix allows us to dispatch on the "kind" of ID when parsing out the information in IDs. It also ensures that a topic ID can never conflict with the ID formed from a declaration.

Linking Attributes

The process of linking node/edge attributes is a little more complicated because we need to figure out how to merge attributes that are added in different `.tao` files. We need to do this without having to do any book keeping in memory so that we do not increase the amount of memory that the linking algorithm consumes.

To solve this problem, I developed an algorithm that uses `O(N)` additional memory. Since we assume that `N << V`, this doesn't change the memory characteristics of the linking algorithm.

Problem

- Nodes and edges have attributes & the TA file has a section at the end where we declare them
- Node and edge attributes can be distributed across `.tao` files because you may decide to add attributes at any point while walking the AST in any file
- We need to collect all of the attributes for each node/edge because we need to write them in a single group for each node/edge in the TA file

Inputs

- `N` open file streams for each `.tao` file
  - One `.tao` file is generated for every .cpp file and every .h file
- Suppose that on average, in every `.tao` file, we have `M` nodes/edges that have attributes
  - The `.tao` file has several sections, but for the purposes of this algorithm we'll assume that our file streams start at the attribute sections

- The attribute section is essentially a list of each node/edge in no particular order with its attributes, also in no particular order
- Recall that because we control the `.tao` file format, we can decide how to structure it to make this algorithm optimal (we'll use this later)

Design Goals/Constraints

1. Must be low memory
   - We assume that any single TAGraph generated by walking the AST of a file is not big, but the sum of all of them for an entire analysis is too big to load all at once
   - That means that going through the `.tao` files and collecting every attribute for every node/edge (brute force) is out of the question
2. Should load only the minimal amount from every `.tao` file
   - Disk I/O is very slow and if we have to run through the `.tao` files over and over again, we greatly increase the runtime of our algorithm
   - There is no hard computational complexity requirement. This is as close as we will get to one.
   - We want to be conscious of runtime and not have an `O(N^2)` solution but also not worry as much because memory is the biggest consideration here.

Current Implementation Currently, we go through each of the `.tao` files one at a time. We load a single node/edge's attributes, and write those attributes to the TA file. We continue to do that until that `.tao` file's attributes have all been written. We then move on to the next `.tao` file and repeat.

This only works on the assumption that a node/edge's attributes will be declared at most in one `.tao` file. That will not be the case in any non-trivial run of Rex because attributes can be added at any time to any node while analysing any file. This is an important freedom we want to keep in Rex. Having separate compilation should not limit our abilities to annotate nodes/edges.

Solution Overview The solution I've come up with takes inspiration from the "merge" routine in merge sort. The neat thing about merge is that you can use it to very quickly determine duplicates because each input to merge is already sorted. Instead of going through all the elements of each list, you can just look at the current item for each one and use it to determine what your next output is and whether it is the same as anything else. The merge routine is fast because its inputs are already sorted.

1. Preprocessing Step As mentioned previously, since the `.tao` files can be anything we want, we can change them to make this process more efficient. Instead of writing the node/edge attributes to the `.tao` files in no particular order, we can use what makes "merge" fast: let's sort the lists!

Sorting requires building a list of the node names and a list of the edges and then calling C++'s `sort()` routine. This fits within the memory characteristics that we want because we are assuming that any single TAGraph is not too big. We will do this sorting at the time of writing the `.tao` file, so that we already have the nodes/edges in memory. Sorting them will not use too much more memory than we already are at that point.

Once the attributes are written to the `.tao` file sorted by their node/edge, the actual algorithm to do the merging (i.e. do the "linking") is reasonably straightforward.

2. Linking Step

1. Load a single node/edge and its attributes from each of the `N` input streams for the `.tao` files and put that into a list
   - Since we know how many `.tao` files there are, the storage for this can be pre-allocated
   - This avoids frequent re-allocation as each node/edge is loaded

2. Sort the vector by the node/edge, leaving the attributes as they are
3. Take the "lowest" sorted node/edge and merge its attributes with any duplicate node/edges from the other input streams
    - All duplicates will be adjacent because we have sorted the list
    - In the common case where we have very few duplicates, this will be very fast because we can stop immediately as soon as we have a non-duplicate
    - Merging is just the process of building a hash map of the attribute names to their values (also very fast)
4. Write the node/edge with its merged attributes to the TA file
    - The memory for the attributes is freed/reused afterwards to avoid using more memory than we need to (nothing from an iteration is kept around after it completes)
5. Load up to `N` more nodes/edges from their `.tao` files to replace the ones that were written
    - One a file runs out of attributes, we can just shorten the original list or otherwise store nullptr or something to ignore that slot
6. Repeat starting from step 2 until we are completely out of attributes to write

Assumptions

1. No single `.tao` file has multiple declarations of the attributes for a given node/edge
    - This is true by design because we keep the attributes in a hash map while walking the AST. Duplicates are not possible.
    - We need this assumption because otherwise we would have to load more than a single node/edge in step 1
2. The total number of attributes for a given node/edge is bound by some constant `K` such that `K << M` and `K << N`.
    - This allows us to assume that the process of merging attributes from duplicates is `O(N)` rather than `O(NK)`

Analysis The most important aspect of this is that it avoids loading more than it needs to from the `.tao` files. We need to keep as many things out of memory as possible to allow Rex to scale up to very big projects.

Step 1: Uses `O(NK)` memory to load a single node/edge from each file. Assumption 2 allows us to simplify this to just `O(N)`.

Step 2: We assume that C++'s sorting algorithm is sufficiently fast. Since runtime isn't as important, we won't state its complexity here. Sorting everything once is much faster than sorting again and again and much less complex than trying to keep the list sorted as we insert into it. The sorting process is in-place, so the memory consumption does not change during this step.

Step 3: Since the list is sorted, finding the smallest value is trivial. We may have up to `N` duplicates, however since merging is fairly fast (Assumption 2), this should not matter at all in terms of runtime. This step will allocate `O(K)` additional memory which according to Assumption 2 is really only roughly `O(1)`.

Step 4: This is `O(1)` and uses no additional memory. The memory we had used in the last step to merge to store the attributes can be freed/reused. If Assumption 2 holds, this is another great way to avoid re-allocating `O(K)` memory in every iteration of this algorithm.

Step 5: We reuse the `O(N)` memory we allocated in step 1 for these new nodes/edges

Step 6: We sort again and continue the process. In the common case, only a few of the items will have changed. Since the list will already be almost sorted, we could potentially do something in the future that optimizes for this. That being said, C++'s sort routine is probably already heavily optimized, so we shouldn't worry about that until we need to.

Summary & Conclusions Overall, I think this is a very good solution that meets our design goals without overcomplicating things. We only use `O(N)` memory which means that as long as we don't have a project with as many source files as nodes or edges, we should use a minimal amount of memory.

TA Object Files

It's worthwhile discussing some of the smaller details of the `.tao` file format. This format was heavily optimized to make linking as fast and as memory efficient as possible. Without it, we would not have been able to create such a performant algorithm.

The sections of the `.tao` file are ordered in exactly the order we need to access them. We start with a "metadata" section that is bascially a list of the sizes of each section. We then go through all the nodes so we can build a symbol table. Then, we go through the edges followed by the node/edge attributes. This order is built to correspond to the order you write facts and attributes to a TA file: first the `$INSTANCE` lines for each node, then the tuples for each fact, then a separate section for the node/edge attributes.

The metadata section I mentioned is particularly nice to have because it acts as an "index" that we can use to quickly jump to any section of the `.tao` file. We never need to stop and inspect the contents of the line we're looking at if we don't want to because we already know what we need to know at the beginning of the document.

Prefixing data with metadata is a common occurrence in the `.tao` format and is very common in almost all binary formats developed elsewhere as well. In fact, every string in the `.tao` file is prefixed with its length so we can allocate once and read the entire string without inspecting its contents. No need to branch on every character to check for a quotation mark or semicolon, we already know exactly what we want.

We also do this for arrays of things. Attribute lists are prefixed with the length so we know exactly how many attributes we need to allocate space for and how many we need to read.

One of the things we do during the linking process is "establish" edges using nodes we know that we should keep. Doing this for every single edge Rex generates would be quite wasteful because there are a lot of edges that we generate that we know are already established because their nodes come from within the compilation unit. That is why the `.tao` file has two sections of edges: unestablished edges and established edges. This allows us to unconditionally write all of the established edges to the TA file without having to check anything for each edge. The insight here is that there is no need to re-establish what we already know to be established.

Finally, we also sort the nodes and edges with attributes because of the reasons described in the "Linking Attributes" section.

Linking: Graph Database

Though we did not get to actually finish the implementation of linking that targets a graph database, I wanted to capture some loose thoughts about the process that may help future developers get started.

> **Aside:** Why linking? Why not write to the graph database during walking? There are two main reasons: 1) performance and 2) atomicity. Querying and writing to a database during walking would slow us down enormously. Our queries are very simple and the data we generate during walking is as well. Another (arguably more important) reason is atomicity. If the walker segfaults or if there is some bug, we want to be able to recover from that failure. Having the walker write to a separate TA Object File and then writing all of that to the database at once gives us the ability to do that. You won't lose hours of collected data just because of one file.

One of the major decisions that you'll make when starting to designing a linking process for the graph database is whether or not your changes should be atomic. That is, if you link together `.tao` files into a graph database, should the second time you link overwrite all of the data or only modify the data that is

already there. Performing operations in a transaction will greatly limit the flexibility of the queries and insertions you can make.

A basic linking algorithm that targets the graph database could be created by just taking the current linking algorithm and replacing the writes to the output TA file with insertions into the database. You would no longer need to worry about making more than 2 passes over the `.tao` files because there is no specific section order like there is with TA. All the nodes, edges, and attributes from each `.tao` file can be inserted at once after you've built the symbol table.

A further extension to explore is whether the symbol table itself can be made superfluous thanks to the database. If you can query uncomitted data during a transaction, this may be possible. I question whether this would be performant enough to be worth it.

A big benefit of using a graph database is that we are no longer constrained by the limitations of the TA format. We can support multiple of the same type of edge where we need to. There are certain edges which semantically can be duplicated with no issues. To implement this, we would need to go through each of our edge types to determine which of them should only be allowed once for a given pair of nodes and which of them should be allowed multiple times.

Incremental Compilation

Separate compilation and linking enable us to do a naive form of incremental compilation where we only re-analyse files that have changed. This feature is common in most compilers and enables programs like `make` to only rebuild parts of projects that need to be built.

A more advanced form of incremental compilation is when a program is able to see the differences between the current form of a compilation unit and its previous form. It can then intelligently decide which facts need to be updated.

Even more sophisticated is the concept of incremental linking. With incremental linking, you can take either the output of the linking process or other separately outputted intermediate files and perform the linking process without having to re-link everything from the original object files.

To implement incremental compilation + linking for Rex, you will likely need to annotate nodes and edges with attributes that relate to where they came from. You can then use this information to only update the facts related to the files that have changed.

Make sure you take advantage of the `.tao` file format being Rex-specific and make any modifications you need to in order to have the metadata you need for incremental compilation.

Good luck! Hope you have fun working on Rex!

Apr 10, 2019

This is a continuation of the task I recorded on Jan 15, 2019. Now that sufficient work has been done on parameter passing, we can address one of the aspects of that task that was left unsolved.

The main problem that we were trying to solve back then was that we needed writes to object member variables to actually reflect that they were written to a particular instance of that object, not a static member on that object. This was an important improvement to the accurancy of Rex's analysis.

We specifically addressed the case where a public member variable was being written through the variable that represents a particular instance. Let's suppose that we have the code:

```
int main() {
  A x;
```

```
  int q = 2;
  x.bar = q;
}
```

The previously completed task performed the relatively easy association between the write to `bar` and its particular instance `x`. Instead of generating `varWrite q A::bar`, we started generating `varWrite q x`. This is "easy" because we have all the information we need immediately when we look at the `x.bar = q` assignment in the AST.

A harder problem to deal with is when we call a method on a class. When we statically analyze the body of a method, we don't know anything about which particular instance of that class is calling the method. The same method body runs for any number of instances of that class. The method may get used in the same file or a completely different one somewhere else, so we can't easily search for its usages either. This creates quite a few complications in our analysis. If we don't model this at all, we have a pretty big blindspot that can greatly affect the accuracy of our results.

To figure this out, suppose we have the following code (annotated with the cases being represented):

```
class Foo {
public:
  static int staticMember = 5;
  int bar;
  int bar2;

  void modifyBar(int x) {
    bar = x;                    // write to member variable
  }

  void modifyBar2(int x2) {
    modifyBar(bar2);            // call to another member function with member variable
    bar2 = x2;                  // write to a second member variable
  }

  static void modifyStatic(int y) {
    staticMember = y;           // write to an actual static member variable
    // Cannot access `bar` or `bar2` at all because this is a static member
  }
};

int main() {
  int w = 22;
  int w2 = 33;

  Foo q1;
  q1.modifyBar(w);              // pass w to member function on instance q1
  q1.bar = w;                   // write w directly to `bar` on instance q1
  q1.modifyBar2(w);             // pass w to *different* member function on the *same* instance q1
  q1.bar2 = w;                  // write w directly to `bar2` on instance q1

  Foo q2;
  q2.modifyBar(w2);             // pass w2 to *same* member function on *different* instance q2

  Foo::modifyStatic(w2);        // pass w2 to *static* member function
}
```

With the work done on parameter passing, this would generate the following `varWrite` facts. Note that the

node IDs here are simplified for the sake of illustration. The actual IDs would be different.

```
class Foo {
public:
  static int staticMember = 5;
  int bar;
  int bar2;

  void modifyBar(int x) {
    bar = x;                          // varWrite x Foo::bar
  }

  void modifyBar2(int x2) {
    modifyBar(bar2);                  // varWrite Foo::bar2 x
    bar2 = x2;                        // varWrite x2 Foo::bar2
  }

  static void modifyStatic(int y) {
    staticMember = y;                 // varWrite y Foo::staticMember
  }
};

int main() {
  int w = 22;
  int w2 = 33;

  Foo q1;
  q1.modifyBar(w);                    // varWrite w x
  q1.bar = w;                         // varWrite w q1
  q1.modifyBar2(w);                   // varWrite w x2
  q1.bar2 = w;                        // varWrite w q1

  Foo q2;
  q2.modifyBar(w2);                   // varWrite w2 x

  Foo::modifyStatic(w2);              // varWrite w2 y
}
```

Notice that the facts generated for the member functions in the class treat the member variables as if they are static. When we analyse member function declarations, we don't have any information about their specific instances, so we can't specialize the facts to each instance like we did above with `x.bar = q`.

The facts generated for the member function calls (e.g. `varWrite Foo::bar2 x` and `varWrite w x`) are from the parameter passing work. I'm not sure if that was fully implemented for member function calls or not, but if it wasn't, it should be added to be consistent with non-member function calls.

The main issue with these facts is that even though the `q1.modifyBar(w)` line and the `q1.bar = w` line result in the same operation being performed, they don't produce edges that can be shown to be equivalent. For `q1.modifyBar(w)`, we transitively see `varWrite w Foo::bar` whereas for `q1.bar = w`, we have `varWrite w q1`.

Ideally, we would like both of those lines to transitively result in an edge from `w` to `q1` so that the work I recorded on Feb 11, 2019 works as expected.

In developing a solution for this, we (Jo and Sunjay) came up with three particular solutions:

1. Inlining Facts

2. Class Context Writes
3. **Function Context Writes**

Each of these is outlined in detail below.

In the end, Jo and I decided to attempt approach (3) even though it shares many of the downsides of approach (2) and is only a little bit more precise. All three of these approaches are a major improvement over the current situation (described above) and each have their own trade-offs.

A key thing to understand before reading about these approachs is that there is no (feasible) perfect way to model this statically. If we aren't actually running the program, it is very hard and even impossible in some cases to accurately describe the entire flow of information in a reasonable amount of time and memory. Enumerating all possible pathways for a given value is completely infeasible and anything less is often inaccurate. Each approach below has trade-offs that either result in infeasibility or cause more inaccuracies. We have chosen an approach that is as best of an approximation as we can come up with given that we need something that runs in an amount of time/memory that is appropriate for our use cases.

1. Inlining Facts

This is the brute force way of solving the problem. The idea is that we collect facts for member functions when we analyze them and also collect the member function calls we see throughout the code. Then we somehow "copy" (or "inline") the facts for each member function at the call site using the particular instance that was called. We can do this recursively to make sure we accurately model member function calls within member function calls. As an example of this, we could take the call to `q1.modifyBar(w)` and copy all of the facts from the original class in order to generate the following facts:

```
class Foo {
public:
  static int staticMember = 5;
  int bar;
  int bar2;

  void modifyBar(int x) {
    bar = x;                          // varWrite x q1
  }

  void modifyBar2(int x2) {
    modifyBar(bar2);                  // varWrite q1 x
    bar2 = x2;                        // varWrite x2 q1
  }

  static void modifyStatic(int y) {
    staticMember = y;                 // varWrite y Foo::staticMember
  }
};
```

Notice that all the references in the facts to a member of `Foo` have been replaced with `q1`, the specific instance that the member function was called on. In reality you would probably only do the inlining for facts related to `modifyBar`. I've shown all the substitutions here just so you get a sense of how that would look in all cases. The facts having to do with the static member `Foo::staticMember` are left the same.

This new set of facts would accomplish our original goal. Given the facts we originally had in `main`, we can now transitively deduce `varWrite w q1`.

```
(varWrite w x) o (varWrite x q1)
  == (varWrite w q1)
```

Of all the methods presented in this log entry, inlining produces the most accurate results because it perfectly models the specific instance being modified by its member function call. It's as if the function body itself was inlined at the call site.

That being said, there are a number of issues with this strategy:

1. **Fact explosion** - You can imagine that for a non-trivial program, this inlining strategy would quickly result in a huge combinatorial explosion of facts. It's not enough to inline just the facts for a single method. Adopting this approach at all would mean that we would need to *recursively* inline facts for method calls within other method calls as well. At that point, we're essentially simulating an actual run of the program.
2. **Order dependence / Additional passes** - One way to implement this is to force Rex to first analyse class declarations and method implementations so we can collect their facts and inline member function calls as we see them. This is very undeseriable and would greatly change the operation of Rex. We want to avoid having to analyse files in any specific order. Another way to implement this is to do an additional pass: first we collect all the facts and all the method calls, then we go back and do the inlining for all the collected method calls. This also fundamentally changes the operation of Rex. Neither of these approaches helps in any way to address the previous concern about a "fact explosion".

Both of these drawbacks combined make this approach almost entirely infeasible. Implementing this would increase the time and memory complexity of Rex in ways that are completely undeseriable.

All is not lost. The fact that this could be solved by a second pass does lead us to some approaches that might work. Whenever we've encountered something like this in the past, we've managed to work around it by generating facts and then using relational composition in the grok scripts to combine them. See the work recorded on Feb 11, 2019 for how we added the `pubVar` and `pubTarget` edges to avoid having a second pass in Rex. If you think about it, the "inlining" we're proposing to do in a second pass is really just a relational operation that we brute forcing to generate a bunch of facts. We should leave relational algebra to grok and avoid doing that ourselves as much as possible.

2. Class Context Writes

So how do we do this without inlining? Well consider that every member function has a `this` context that it mutates. Any write to a member variable is a write to `this`. That means that if we consider `this` to be the "instance" of the class being written to, we can do the same thing as what we did with public member variables (recorded on Jan 15, 2019). Every write to a member of `this` can be recorded as a write to `this`.

```
class Foo {
public:
  static int staticMember = 5;
  int bar;
  int bar2;

  void modifyBar(int x) {
    bar = x;                          // varWrite x Foo::this
  }

  void modifyBar2(int x2) {
    modifyBar(bar2);                  // varWrite Foo::this x
    bar2 = x2;                        // varWrite x2 Foo::this
  }

  static void modifyStatic(int y) {
    staticMember = y;                 // varWrite y Foo::staticMember
  }
};
```

Notice how similar this is to the substitutions we did in the inlining example. Instead of the specific instance `q1`, we've generalized to all instances with `Foo::this`. (Small detail: We need to use `Foo::this` because just using the ID `this` would not be unique across all declared classes.)

The change being proposed here is not a substitution that happens in a separate pass. This changes the walker to generate facts for a new variable node `Foo::this`. These facts would replace the ones we were previously generating with specific members (e.g. `Foo::bar`, etc.). (See the work recorded on Jan 15, 2019 for more justification for just using the specific instance and not the specific field.)

Does this solve the original issue we're trying to tackle? Well not directly. There still isn't any way to transitively get `varWrite w q1` from `q1.modifyBar(w)`. To finish this solution, we also need to generate an additional fact with every (non-static) method call.

```
int main() {
  int w = 22;
  int w2 = 33;

  Foo q1;
  q1.modifyBar(w);                  // varWrite Foo::this q1
                                    // varWrite w x

  q1.bar = w;                       // varWrite w q1

  q1.modifyBar2(w);                 // varWrite Foo::this q1
                                    // varWrite w x2

  q1.bar2 = w;                      // varWrite w q1

  Foo q2;
  q2.modifyBar(w2);                 // varWrite Foo::this q2
                                    // varWrite w2 x

  Foo::modifyStatic(w2);            // varWrite w2 y
}
```

Notice the new facts `varWrite Foo::this q1` and `varWrite Foo::this q2`. These model that because we are calling a member function, that member function can *possibly* write to that particular instance. With these facts, we now have enough to transitively get to the fact that we want:

```
(varWrite w x) o (varWrite x Foo::this) o (varWrite Foo::this q1)
  == (varWrite w q1)
```

A very natural way to read this is as a three step process:

1. The variable `w` is written to the parameter `x`
2. The parameter `x` is written to the `this` context of `Foo`
3. Changes to the `this` context of `Foo` are applied to the instance `q1`

This approach even works with nested method calls because we can generate `varWrite Foo::this Foo::this` whenever we see a method call another method.

This whole approach works quite well! That being said, it also comes with its own downsides. Now you get to see why this code example was crafted with the specific cases above. :)

One of the downsides of this approach is that it isn't very precise. As soon as we add more than one instance, we can use the generated facts to produce incorrect conclusions. Consider the following valid line of reasoning created by only slightly modifying the previous proof:

```
(varWrite w x) o (varWrite x Foo::this) o (varWrite Foo::this q2)
```

```
  == (varWrite w q2)
```

Even though the instance `q2` is never written to by the variable `w`, we're able to deduce that it is because it generates a fact `varWrite Foo::this q2`. We need to generate that fact because of the line `q2.modifyBar(w2)`. In a similar vein, we can also deduce `varWrite w2 q1` even though there is no such mutation.

This is what I would call a "necessary trade-off". We can't be fully accurate because we're doing static analysis and never actually running the program. As hard as we might try, we don't actually know the order functions will be called. That being said, we can get "close enough" and model the calls that can occur while also accepting that there will be false positives.

The third and final approach described below is a slight refinement of this method that still shares the same inaccuracies, but eliminates a few of the cases where they can occur.

3. Function Context Writes

This refinement of the previous idea was originally proposed by Jo. This improves the results by having a specific `this` context for each function rather than the entire class. It doesn't get rid of all false positives, but it will eliminate some of them.

The idea is that we change the generated facts to look as follows:

```
class Foo {
public:
  static int staticMember = 5;
  int bar;
  int bar2;

  void modifyBar(int x) {
    bar = x;                          // varWrite x Foo::modifyBar::this
  }

  void modifyBar2(int x2) {
    modifyBar(bar2);                  // varWrite Foo::modifyBar::this Foo::modifyBar2::this
                                      // varWrite Foo::modifyBar2::this x

    bar2 = x2;                        // varWrite x2 Foo::modifyBar2::this
  }

  static void modifyStatic(int y) {
    staticMember = y;                 // varWrite y Foo::staticMember
  }
};

int main() {
  int w = 22;
  int w2 = 33;

  Foo q1;
  q1.modifyBar(w);                    // varWrite Foo::modifyBar::this q1
                                      // varWrite w x

  q1.bar = w;                         // varWrite w q1

  q1.modifyBar2(w);                   // varWrite Foo::modifyBar2::this q1
                                      // varWrite w x2
```

```
  q1.bar2 = w;                      // varWrite w q1

  Foo q2;
  q2.modifyBar(w2);                 // varWrite Foo::modifyBar::this q2
                                    // varWrite w2 x

  Foo::modifyStatic(w2);            // varWrite w2 y
}
```

With these changes, every method call is associated with that particular method. If two instances call a disjoint set of methods, their generated facts won't interact. As mentioned, this doesn't solve all the problems, but it does slightly improve the accuracy of the results.

Implementation Details

A few key details to note when implementing this solution:

You need to actually change the way facts are being generated in Rex for method declarations. None of this involves a doing any sort of substitution later.

Make sure that the node ID you use is not `Foo` or simply the word `this`. The node ID needs to be specific to the class *and* the method that you are generating facts for. We used `<class name>::<method name>::this` in the examples above. C++ does not allow `this` to be used in any other context other than as the function context, so that should be unique.

Member variables may show up as either `this->memVar` or `memVar`. Clang's AST probably provides the same AST node for both.

This change applies only to member variables, not static variables in a class

You should only modify the facts for member functions, not static functions in a class. Member functions may access static variables and static functions cannot access member variables.

Feb 11, 2019

Merge Request: https://git.uwaterloo.ca/swag/Rex/merge_requests/7

The task I was addressing:

> This task is meant to improve the precision of our Behaviour Alteration interactions. The ultimate goal is the track whether some change to a variable in one component (say component A) causes another component (say component B) to behave differently. What the analysis is supposed to do is find all variable assignments, and for each variable assignment (say x = y), the analysis should trace from this statement all possible paths through the program in which the data assigned in VA (i.e., x) is used:
>
> - if x is used in the assignment expression of another variable z, then we care how z is used
> - if x is passed in a parameter in a function call, then we care how the corresponding formal parameter is used within the function
> - if x is passed in a data field in a ROS message, then we care in the receiving components about any formal parameter in any callback function that uses that data field
> - eventually, we care whether a decision condition (e.g., the condition of an if statement) refers to the value of x

> I think that a lot of short cuts were taken in computing Behaviour Alteration interactions; I'm not even sure that we know about all of them.
>
> For starters, the analysis did not consider the passing of parameters or the contents of ROS messages. I think what the current analysis does is simply assume that if x is in component A, and A sends a message to component B, then we need to assume that it is possible that x has passed to B. I don't even know what the analysis starts following in B — all of the parameters in the call back function?
>
> So, I think there are a number of sub tasks to tackle.
>
> 1) Extract facts about formal and actual parameters of functions.
> 2) Modify the Behaviour Alteration interaction script to consider the flow of information from one function A to another function B only if there is a call to function B where one of the actual parameters is a "variable of interest"
> 3) Restrict the "variables of interest" in the new function to the formal parameters that are associated with the actual parameter that was of interest in the calling function.
>
> We'll need to do the same thing with data fields in ROS messages.
>
> 4) Extract facts about data fields in a sent ROS message, and data fields in a received ROS message.
> 5) Modify the Behaviour Alteration interaction script to consider the flow of information from one function to another only if the sent message includes a data field whose value was computed from a variable of interest.
> 6) Restrict the "variables of interest" in the receiving component to be the formal parameters of the callback function triggered by the ROS message.
>
> There is still going to be lots of imprecision, because we are looking for possible flows of information, and it might turn out that at run time, some of these flows are not possible, or are not always taken (because they are conditional on the values of other variables). But I think that the analysis could be more precise than it currently is.

I was specifically working on 4, 5, and 6 (the tasks specific to ROS messages).

Background

ROS message passing code typically has the structure shown below.

For a **Publisher**:

```
NodeHandle nh = ...;
Publisher topic = nh.advertise<MyMess>("topicname", 1000); // (1)

// ...

MyMess mess; // (2)
mess.field1 = used_in_control_flow;
mess.field2 = 2;
mess.field3 = "To be or not to be, that is the question?";

topic.publish(mess); // (3)
```

The call in (1) registers the topic "topicname" with a queue size of 1000. The returned `Publisher` instance can be used to publish messages for this specific topic. The type `MyMess` provided to this function is the

message type for this topic. An instance of this type will be provided as an argument to each callback that subscribes to this topic.

In (2), you see the code typically used to setup a message for publishing. Notice that the message is populated field by field, this is key because it means that we need the changes I made on Jan 15th (see below) in order for this to work. If we can't track that `used_in_control_flow` was assigned to `mess`, we won't be able to accurately account for data-flow from this message to the subscriber.

The call in (3) takes the variable `mess` and sends it to each subscriber. This concept will become very important later, so keep this function in mind.

For a **Subscriber**:

```
NodeHandle nh = ...;
Subscriber sub = nh.subscribe<MyMess>("topicname", 1000, &Foo::callback); // (1)

// ...

void Foo::callback(const MyMess::ConstPtr& received) { // (2)
    this->member_var = *received; // (3)
    // ...
}
```

The call in (1) registers the callback `Foo::callback` as a subscriber for the topic "topicname" with queue size 1000. The type `MyMess` is the message type expected from the queue. The callback is passed to `subscribe` as a function pointer.

The callback declared in (2) takes a const reference to the message type. Most ROS subscribers actually take a special `ConstPtr` version of the message type. Since we don't do any type checking, this doesn't tend to affect our analysis of the data-flow. All we care about is that data went from variable A to variable B. The types are immaterial.

As seen in (3), a key consequence of most callbacks taking a `ConstPtr` argument is that the first operation in the callback is usually to dereference the received value. This is the case in autonomoose code and in other ROS code I found online.

Solution Walkthrough

The key insight about the ROS publish/subscribe model that drives the rest of this implementation is that by calling `publish`, we are indirectly calling the callbacks registered with `subscribe`. Not only that, but we are also writing the variable passed to `publish` to the first argument of each callback. This is **equivalent** to if the call to publish had been replaced by direct calls to each callback registered for that topic. That means that we can model a `publish` in the **same** way that we model a function call.

We already track the *call* of the callback through the existing `publish` and `subscribe` relations that were already present in Rex. The novel part of this work is that we will now be able to see the *data-flow* from the fields of the message to the parameter of the callback. This is explained in detail below.

This work goes hand-in-hand with the work done on parameter passing. This feature combined with the parameter passing work gives us a complete picture of the data-flow through the variables, parameters, and messages of a ROS program.

To understand the solution and how it works, it is helpful to look at the `varWrite` facts extracted from the publish and subscribe examples above.

For the publisher code, we would get:

```
varWrite used_in_control_flow mess
```

For the subscriber code, we would get:

```
varWrite received Foo::member_var
```

> **Note:** Before I made my changes, the `varWrite` fact for the subscriber case was not being generated. This is because ROS previously did not treat the dereference operator as a "read". That meant that any assignment `a = *b` was not registered as `varWrite b a`. One of the decisions made as part of this work is to start to treat a dereference as a read so that this approach can actually work.

Notice that there is nothing connecting the write from the control flow variable `used_in_control_flow` to the `received` parameter of the callback. This is despite the fact that we can see that this write will occur when the message is published.

To solve this, let's add another fact to connect the published variable to the parameter of the callback. Whenever we extract a call to `publish`, we can add a `varWrite` fact that represents the message variable being written to the callback's parameter:

```
varWrite mess received
```

With this, we would have the following 3 facts:

```
varWrite used_in_control_flow mess
varWrite received Foo::member_var
varWrite mess received
```

With transitive closure, it is easy to see that we can use these facts to know that the control flow variable is eventually written to `member_var`:

```
(varWrite used_in_control_flow mess) o (varWrite mess received)
  == (varWrite used_in_control_flow received)

(varWrite used_in_control_flow received) o (varWrite received Foo::member_var)
  == (varWrite used_in_control_flow Foo::member_var)
```

This `varWrite used_in_control_flow Foo::member_var` fact completes our goal of modelling the data-flow from publishers to subscribers. We can see in this fact that this approach would accurately capture that a control flow variable was written to `member_var` through an indirect call to the callback.

Connection to Parameter Passing

The parameter passing work (done separately from all of this) models the passing of parameters to functions in a similar way to what is presented here. Much like how we model a publish as a variable being written to the parameter of a function, the parameter passing work generalizes this idea by modelling a function call as each actual parameter being written to each formal parameter.

The result is that both approaches are completely compatible. Both my approach to ROS publish/subscribe and this approach to parameter passing end up with edges pointing from variables to the parameters they are passed to.

This gives us a way to model the entire data-flow through the program just using `varWrite`.

(A slight caveat to this point is discussed in the Implementation section below. While the point still holds, the implementation makes it slightly more complicated than what is described here.)

The outcome of this is that we can modify the behaviour alteration scripts to use only `varWrite` *and* be more precise because we are only looking at the direct data-flow now, not the other information about calls/writes/etc.

```
masterRel = varWrite + call + write + varInfFunc; //Before
masterRel = varWrite; // After
```

(Again, the actual implementation complicates this a bit. I'm using this slight simplification to make sure the rationale for these changes is clear.)

Implementation

To actually implement these changes, we need to slightly modify the approach presented so far. The problem is that we can't generate `varWrite` edges from publish calls to their subscribers directly because we don't necessarily know about every subscriber when we encounter the call to `publish`.

One (undeseriable) was to fix this would be to make Rex do two passes. One pass would collect all of the calls to publish and all of the callbacks passed to subscribe. The second pass would join the variables passed to publish to the callback parameters. This is overly complicated and involves adding another pass to Rex which is extremely undeseriable compared to a method that only uses one pass.

Luckily, there is a way to do this in one pass. This approach is much simpler and more akin to what we already do in Rex in cases where we don't have all of the information we need when we are walking through the AST the first time.

To model this `varWrite`, we can add two new edge types:

1. `pubVar messVar topic` - Represents that the variable `messVar` will be published (sent) to any subscribers to `topic`.
2. `pubTarget topic param` - Represents that the callback parameter `param` is the target of a publish for `topic`. (The use of the word "target" here refers to the fact that `publish(messVar)` will "target" this variable to write `messVar`.)

Each of these edges can be added independently of how many publishers or subscribers we have seen so far. If we use the relational composition operator on these relations, we get an edge that is equivalent to the `varWrite` edge we were aiming for.

The facts we were previously aiming for were:

```
varWrite used_in_control_flow mess
varWrite received Foo::member_var
varWrite mess received
```

With this implementation, we would instead get:

```
varWrite used_in_control_flow mess
varWrite received Foo::member_var
pubVar mess "topicname"
pubTarget "topicname" received
```

You can see that `pubVar o pubTarget` gives us the tuple `(mess, received)`. That means that this approach can give us facts that are equivalent to the facts we were aiming for.

The "caveat" I was referring to before is that by having these two new edge types, the changes to the script aren't as simple as "just use `varWrite`". We have to go slightly further than that and use both `varWrite` and `pubVar o pubTarget` together to model the entire data-flow of the program.

```
masterRel = varWrite; // In theory

publishWrites = pubVar o pubTarget; // As implemented
masterRel = varWrite + publishWrites;
```

Final Observations

This approach successfully models data-flow from publishers to subscribers in ROS, but the facts extracted are only meaningful if we analyse multiple components together at the same time.

For example, the autonomoose project is divided up into many different components. Each component may publish some messages and subscribe to others, but it is very rare for the same component to publish and subscribe to the same topic. In order to analyse behaviour alterations caused by ROS messages, we would have to make sure we analyse all of the components that publish/subscribe to topics together. You can check that the script is actually looking at these interactions by printing the value of `publishWrites` and ensuring that it is actually not empty.

Jan 15, 2019

Merge Request: https://git.uwaterloo.ca/swag/Rex/merge_requests/5

The task I was addressing:

> Fix Rex's extraction of object member variables. Currently treats all member variables as static (Foo foo1; Foo foo2; foo1.x = 5; foo2.x = 10; are treated as writes to the same variable, Foo::x).

Background

Suppose we have the following C++ code:

```
class Foo {
public:
    int bar;
};

int main() {
  Foo foo1;
  Foo foo2;
  Foo foo3;
  int used_in_control_flow = getfromconfig();

  foo1.bar = used_in_control_flow;

  if (used_in_control_flow > 2) {
    foo2.bar = foo1.bar;
  }

  foo3 = foo2;
}
```

Before my changes, this would generate the following `varWrite` facts:

```
varWrite used_in_control_flow Foo::bar
varWrite Foo::bar Foo::bar
varWrite foo2 foo3
```

The problem here is that by using `Foo::bar`, we don't adequately capture that the values being written are *specific instances* of `Foo`, not some static property on the `Foo` class itself. You can see this clearly in the

`varWrite Foo::bar Foo::bar` fact. This doesn't represent anything close to what is really happening. It would be nice if we could use Rex to talk about specific instances, not just the entire class itself.

Solution Walkthrough

Initially, we thought that it would make sense to change Rex's behaviour to take the field being written into account. That means that instead of generating `varWrite Foo::bar Foo::bar`, we would get something similar to `varWrite foo1.bar foo2.bar` instead.

This seems like it would work because we have both the specific instance information, and the field as well. The problem however is that it still doesn't allow us to completely model the flow of data through the variables. Let's see what the set of facts would look like with this change:

```
varWrite used_in_control_flow foo1.bar
varWrite foo1.bar foo2.bar
varWrite foo2 foo3
```

Notice that even though we know that the value of `used_in_control_flow` was copied into `foo3` indirectly, we can't see that through these facts. The atom `foo2.bar` is not the same as `foo2`, so we can't use the transitive closure operation to join the two.

After much discussion, Dr. Jo Atlee came up with a better approach. This is a "cruder" way of looking at the writes to the fields of an instance. Rather than consider which specific field was written, we can consider each assignment as a write to the entire instance instead. So instead of `varWrite foo1.bar foo2.bar`, we now get `varWrite foo1 foo2`.

Let's see how the facts look with this approach:

```
varWrite used_in_control_flow foo1
varWrite foo1 foo2
varWrite foo2 foo3
```

Notice that even though there is technically less information captured in these facts, the connection between them can be seen clearly. Thanks to transitivity, we can get a clear path to see that `foo3` is written with the value of the variable `used_in_control_flow`.

In pseudo-grok notation:

```
(varWrite used_in_control_flow foo1) o (varWrite foo1 foo2)
  == (varWrite used_in_control_flow foo2)

(varWrite used_in_control_flow foo2) o (varWrite foo2 foo3)
  == (varWrite used_in_control_flow foo3)
```

This final `varWrite used_in_control_flow foo3` fact shows that we have accomplished our goal of making Rex's analysis of public member variables more precise in order to better model the flow of data through the program. We can now see that `foo3` is written to by a control flow variable even though that write is performed indirectly through the fields of another instance.