Empirical analysis and optimization of an *NP*-hard algorithm using CSP and FDR

Douglas A. Creager, Andrew C. Simpson

[1]Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD
United Kingdom

Abstract. *In many cases where an algorithm is provably NP-hard, this intractability is a worst-case bound that only applies to pathological inputs. In these cases, by exploiting knowledge of the specific structure of "real-world" inputs, the algorithm can be shown to be much more efficient in the "normal" case. However, when studying a new problem, this can be hard to show if it is not obvious which structural constraints exist, and which ones would lead to increases in efficiency. In this paper, we show how one can describe the underlying problem declaratively as a CSP process and use the FDR refinement checker to explore the complexity space of the problem. By knowing which optimizations FDR uses to find solutions more efficiently, we can determine under which conditions the worst-case intractable algorithm executes efficiently, and incorporate analogous optimizations into the algorithm to exploit these conditions.*

1. Introduction

The class *NP* is often used as a benchmark for deciding when an algorithm or problem is "difficult"; whereas the space requirements and running time of an algorithm with a polynomial solution will increase reasonably with the size of the input, *NP* space requirements and running times tend to scale exponentially. Algorithms in *NP* are often therefore only realistically useful on the most trivial of inputs.

Of course, the fact that a particular algorithm is in *NP* does not necessarily imply that the underlying problem is itself difficult — it might be that a polynomial-time algorithm exists but has not yet been discovered. *NP*-hard algorithms are those where this is highly unlikely; any polynomial-time solution to an *NP*-hard problem could be used to create polynomial-time solutions to *every* other algorithm in *NP*. Though it has not been proved, our current intuition is that $P \neq NP$, and therefore that *NP*-hard problems cannot have any polynomial-time solutions.

Sometimes, however, this is only a worst-case bound on the complexity of an algorithm. There might be classes of inputs for which the problem simplifies and becomes tractable. For example, this is exactly the case with Petri nets [Petri 1962, Petri 1963]: in the general case, the reachability problem is *EXPSPACE*-complete. However, as summarized in [Esparza 1998], by introducing constraints on the structure of the Petri net, different classes are formed with tractable reachability algorithms. If we can guarantee that each place in the Petri net will only ever contain at most one token, then reachability becomes *PSPACE*-complete. If, in addition, the net contains no cycles, reachability becomes *NP*-hard. If we further stipulate that each place in the Petri net can have at most one output transition, reachability becomes polynomial.

When studying a new problem domain, an important task is to discover which of the associated algorithms are *NP*-hard, and if possible, which simplifications of the problem domain make these algorithms tractable in the "normal" case. Finding these subproblems is not always trivial; often, a lot of analysis and intuition is needed before an appropriate breakthrough is made.

In this paper we present an alternative, empirical, approach, applying it to a new data transformation discovery algorithm that is provably *NP*-hard in the worst case. We first create a description of the problem using the Communicating Sequential Processes process algebra (CSP) [Hoare 1985, Roscoe 1998]. This mathematically rigorous and machine-readable problem description can then be solved by the FDR refinement checker [Roscoe 1994, Scattergood 1998]. By using many varying inputs and slight modifications to the process description, FDR can be used to analyze the space and time complexity of different algorithmic solutions to the problem. With an understanding of the particular normalizations and optimizations that FDR uses "under the hood", we can then use the same strategies when developing our own algorithmic solution.

There are many other examples in the literature of empirical approaches to analyzing the complexity of an algorithm or program (including, but not limited to, [Jones 1986, Breese et al. 1998, Hunt et al. 1998]). However, these approaches focus on existing low-level implementations of an algorithm. Our approach differs from these examples in that we examine a high-level description of the *problem*, written in a declarative style. This allows us to identify useful optimizations and an overall implementation strategy *before* developing a low-level algorithm to solve the problem.

The rest of this paper is organized as follows. Section 2 presents an overview of our particular problem of interest. Section 3 shows how the problem can be formulated as a refinement between two CSP processes. Section 4 shows how FDR can be used to analyze the space and time complexity of the problem, and how different changes to the CSP script affect the complexity of the solution. In Section 5, we interpret these measurements in terms of real-world transformation graphs, and use these insights to develop a useful algorithmic solution. Finally, Section 6 discusses our results and suggests areas for future research.

2. Problem description

The problem that we consider involves the automated discovery of data transformations. In a heterogeneous environment like the Internet, communicating applications will likely encode and structure their data differently, even if the data represents logically similar real-world concepts and objects. Since rewriting the applications to use an identical data model will often be an infeasible solution, some form of *data transformation* is needed to reconcile these differences. Ideally, the discovery of these transformations would be automated, reducing the amount of effort needed to link two disparate applications.

The approach taken in [Creager and Simpson 2006] is to exploit the fact that transformations are composable. We assume that some *atomic transformations* will necessarily be written manually; however, given a sufficient number of them, a directed graph can be constructed with datatypes as nodes and atomic transformations as edges. An example transformation graph, containing datatypes and transformations for a variety of microscope image formats and their associated metadata, is shown in Figure 1. Since atomic

transformations are composable, paths in the graph represent executable *compound transformations*, and an efficient pathfinding algorithm, such as Dijkstra's [Dijkstra 1959] or Bellman-Ford [Bellman 1958, Ford and Fulkerson 1962], can be used to discover them. Further, the same graph can be used to support use cases with different non-functional requirements through the appropriate use of edge weights.

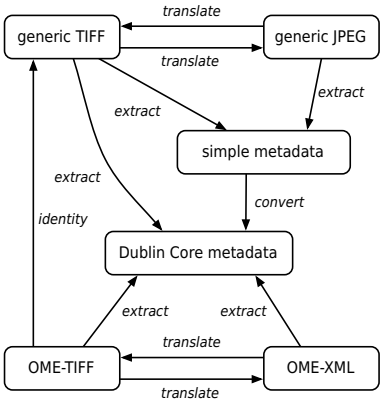

Figure 1. A transformation graph for images and associated metadata

Unfortunately, since graph edges have exactly one source node and exactly one sink node, this graph-based model limits transformations to a single input and output. If we want to support transformations of higher arity, a more complex model is needed. An obvious extension is to use *hypergraphs* [Berge 1973, Berge 1989, Ausiello et al. 1983], whose *hyperedges* can connect multiple nodes. Figure 2 shows how a polyadic transformation graph could be represented as a hypergraph. The analogous *shortest hyperpath* algorithm would then be used to discover compound transformations. Hyperpaths are more complex than standard paths, in that there are a number of ways to calculate a hyperpath's weight given the weights of its constituent edges [Italiano and Nanni 1989, Ausiello et al. 1992]. For instance, if an edge appears in a hyperpath multiple times, its weight can either be counted exactly once, or once for each of its appearances in the hyperpath. These different metrics yield different complexities for the shortest hyperpath problem, some of which are polynomial; however, the metric that we would use for transformation discovery, *cost*, is *NP*-hard.

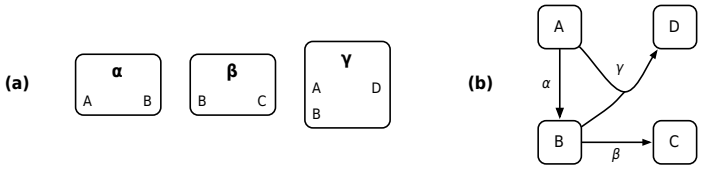

Figure 2. A polyadic transformation graph represented by a hypergraph

The hope, however, is that this is a worst-case bound for pathological transformation graphs, and that for more common "real-world" graphs, the discovery algorithm would be tractable. In the remainder of this paper, we use CSP and FDR to investigate this hypothesis.

3. CSP implementation

In this section, we describe a prototype implementation of the transformation discovery problem described in the previous section, written in CSP. The usual strategy for working with CSP specifications is to define two processes: one providing a specification of what the system should do, and the other describing a particular implementation of the system. One then uses a refinement checker such as FDR to verify that the implementation refines, and therefore satisfies, the specification.

For our transformation graph problem, we will use the first process to describe the structure of the graph, and the basic rules about when transformations can be executed. We will use the second process to describe the specific property that we are looking for: that, given instances of a particular set of *initial datatypes*, there is some sequence of transformations that can be executed that will yield instances of a different set of *desired datatypes*. Note that we do not describe *how* to find solutions; we only provide a declarative description of the problem structure and of valid solutions.

3.1. Graph structure

We start by declaring the CSP types needed for the specification. The *Datatype* type represents a single datatype from the transformation graph. (The overloading of the term "datatype" is unfortunate but unavoidable; we will use "type" to refer to the syntactic concept in the CSP language, and "datatype" to refer to a node in a transformation graph.) A *Transformation* has a unique identifier, and is defined by two sets of datatypes: one for its inputs and one for its outputs. A particular transformation graph can be encoded by providing concrete values for the *Datatype* type and the *Transformations*, *GivenTypes*, and *DesiredTypes* variables. The *Transformations* variable contains all of the transformations in the graph. The *GivenTypes* variable specifies which datatypes we are given instances of, while *DesiredTypes* specifies which datatypes need to be generated by the discovered compound transformation.

$$\textsf{datatype } \mathit{Datatype}, \mathit{XformID}$$
$$\textsf{nametype } \mathit{Transformation} = \mathit{XformID} \times (\mathbb{P}\,\mathit{Datatype}) \times (\mathbb{P}\,\mathit{Datatype})$$
$$\textsf{variable } \mathit{Transformations}, \mathit{GivenTypes}, \mathit{DesiredTypes}$$

Next we define the event channels that will be used in the specification. The *have* channel signals when a datatype has become available, regardless of how it was obtained. The *given* channel is used to notify other processes which datatypes are given. The *execute* channel signals that a particular atomic transformation has executed. The *produce* channel indicates that a datatype has been produced as the output of some transformation. Finally, the *finish* channel signifies that a datatype has been used as the final result of the compound transformation.

$$\textsf{channel } \mathit{given}, \mathit{have}, \mathit{produce}, \mathit{finish} : \mathit{Datatype}$$
$$\textsf{channel } \mathit{execute} : \mathit{XformID}$$

Now we can construct the CSP process that represents the structure and rules of a transformation graph. We follow the standard approach of declaring subprocesses for each of the individual properties or constraints of the system, which we then compose together into a final specification using parallel composition.

We first define a *MakeAvailable* process that is responsible for generating *have* messages whenever a datatype instance becomes available. This can happen in one of two ways: we can be given the instance (in which case we match a *given* message), or it can be generated by the execution of a transformation (in which case we match a *produce* message).

$$
\begin{array}{l}
\alpha(\mathit{MakeAvailable}) = \{\!|\mathit{given}, \mathit{produce}, \mathit{have}|\!\} \\
\mathit{MakeAvailable} = \\
\quad \mathit{given}?t \rightarrow \mathit{have}!t \rightarrow \mathit{MakeAvailable} \\
\quad \Box \\
\quad \mathit{produce}?t \rightarrow \mathit{have}!t \rightarrow \mathit{MakeAvailable}
\end{array}
$$

Next we define a *Given* process that generates the initial *given* messages for the datatypes that we start with. The alphabet of this process contains *all given* messages, even though only certain *given* messages are created; this ensures that CSP events only appear for those datatypes that actually are given to us.

$$
\begin{array}{l}
\alpha(\mathit{Given}) = \{\!|\mathit{given}|\!\} \\
\mathit{Given} = |||\, t : \mathit{GivenTypes} \bullet \mathit{given}!t \rightarrow \mathit{Stop}
\end{array}
$$

Next we define a process to handle the *finish* messages. We keep track of which datatypes are available; when one of the *DesiredTypes* becomes available, we allow at most one a *finish* event for it.

$$
\begin{array}{l}
\alpha(\mathit{Finish}) = \{\!|\mathit{finish}, \mathit{have}|\!\} \\
\mathit{Finish} = \\
\quad \textsf{let} \\
\quad\quad \mathit{Have}(\mathit{avail}, \mathit{finished}) = \\
\quad\quad\quad \mathit{have}?t \rightarrow \mathit{Have}(\mathit{avail} \cup \{t\}, \mathit{finished}) \\
\quad\quad\quad \Box \\
\quad\quad\quad \mathit{finish}?t : (\mathit{avail} \setminus \mathit{finished}) \cap \mathit{DesiredTypes} \rightarrow \\
\quad\quad\quad\quad \mathit{Have}(\mathit{avail}, \mathit{finished} \cup \{t\}) \\
\quad \textsf{within} \\
\quad\quad \mathit{Have}(\varnothing, \varnothing)
\end{array}
$$

Our next process is responsible for preventing a particular transformation from executing before all of its inputs are satisfied. We define it similarly to the *Finish* process: we keep track of which input datatypes are available. Once all of them are available, we allow any number of *execute* events to occur for this transformation.

$$
\begin{array}{l}
\alpha(\mathit{XformPrereq}(\ (\mathit{id}, \mathit{inputTypes}, \mathit{outputTypes})\ )) = \\
\quad \{\mathit{execute}.\mathit{id}\} \cup \{\ t : \mathit{inputTypes} \bullet \mathit{have}.t\ \} \\
\mathit{XformPrereq}(\ (\mathit{id}, \mathit{inputTypes}, \mathit{outputTypes})\ ) = \\
\quad \textsf{let} \\
\quad\quad \mathit{Have}(\mathit{avail}) = \\
\quad\quad\quad (\mathit{avail} = \mathit{inputTypes})\ \&\ \mathit{execute}!\mathit{id} \rightarrow \mathit{Have}(\mathit{avail}) \\
\quad\quad\quad \Box \\
\quad\quad\quad \mathit{have}?t : \mathit{inputTypes} \rightarrow \mathit{Have}(\mathit{avail} \cup \{t\})
\end{array}
$$

within
$Have(\varnothing)$

For the transformation graphs described in this chapter, we assume that every datatype is *reusable*: that any instance of the datatype can be used multiple times without penalty. For this reason, we do not remove any elements from the set of available datatypes in the *Finish* and *XformPrereq* processes. If desired, we could use a more complicated definition for these processes to limit the number of times that a particular datatype could be consumed.

The above process verifies that the prerequisites are satisfied for a single transformation. We must use parallel composition to combine them together: since multiple transformations might be waiting for the same datatype to satisfy an input, they must be notified of its availability simultaneously. This means that they must synchronize on the corresponding *have* event. This parallel composition yields the *Prereqs* process, which verifies the prerequisites of each atomic transformation simultaneously.

$$\alpha(Prereqs) = \bigcup\{\, xf : Transformations \bullet \alpha(XformPrereq(xf)) \,\}$$
$$Prereqs = \|\, xf : Transformations \bullet XformPrereq(xf)$$

Next we define a process that describes what happens when a particular transformation is executed. It waits for the appropriate *execute* event, after which it outputs *produce* events for each output datatype. We use replicated interleaving to allow the *produce* events to occur in any order. The overall process then ends in *Skip*.

$$\alpha(ExecuteOneOnce(\ (id, inputTypes, outputTypes)\ )) =$$
$$\quad \{execute.id\} \cup \{\, t : outputTypes \bullet produce.t \,\}$$
$$ExecuteOneOnce(\ (id, inputTypes, outputTypes)\ ) =$$
$$\quad execute!id \rightarrow (|||\, t : outputTypes \bullet produce!t \rightarrow Skip)$$

The *ExecuteOneOnce* process is parameterized on the definition of a transformation; now we instantiate this process for each of the actual transformations in the graph. The *ExecuteAnyOnce* process allows the environment to execute any one transformation. Its alphabet consists of *all* of the *produce* messages, since we want to prevent datatypes that do not play a part in some transformation from being produced.

$$\alpha(ExecuteAnyOnce) = \{|execute, produce|\}$$
$$ExecuteAnyOnce = \Box\, xf : Transformations \bullet ExecuteOneOnce(xf)$$

With the *ExecuteAnyOnce* process, we have allowed the environment to execute a single transformation. Now we allow it to execute a sequence of them. Since the *ExecuteAnyOnce* process ends with a *Skip* (due to it being defined in terms of *ExecuteOneOnce*), we can accomplish this with a recursive sequential composition. The *Execute* process allows *any* sequence of transformations to be executed; it does not need to take into account whether a transformation has its inputs satisfied, since this constraint is handled by the *Prereqs* process.

$$\alpha(\mathit{Execute}) = \alpha(\mathit{ExecuteAnyOnce})$$
$$\mathit{Execute} = \mathit{ExecuteAnyOnce} \fatsemi \mathit{Execute}$$

Finally, we can merge together all of the previous processes using parallel composition. This yields an overall *Graph* process that satisfies the constraints introduced by each of its constituent parts. We also provide a view of the graph (*GraphOutputs*) that hides everything except for the *finish* channel, since our description of a successful solution will only depend on which final datatypes are produced.

$$\alpha(\mathit{Graph}) = \{|\mathit{given}, \mathit{have}, \mathit{execute}, \mathit{produce}, \mathit{finish}|\}$$
$$\mathit{Graph} = \mathit{MakeAvailable} \parallel \mathit{Given} \parallel \mathit{Finish} \parallel \mathit{Prereqs} \parallel \mathit{Execute}$$

$$\alpha(\mathit{GraphOutputs}) = \{|\mathit{finish}|\}$$
$$\mathit{GraphOutputs} = \mathit{Graph} \setminus (\alpha(\mathit{Graph}) \setminus \{|\mathit{finish}|\})$$

3.2. Transformation discovery process

Next we construct the CSP process that tests whether all of the desired datatypes are eventually produced by some compound transformation. We can do this by constructing an appropriate traces refinement. We construct a Want_T process that allows the appropriate *finish* events to occur in any order:

$$\mathit{Want}_T(\varnothing) = \mathit{Stop}$$
$$\mathit{Want}_T(\mathit{types}) = \Vert\!\Vert\, t : \mathit{types} \bullet \mathit{finish}!t \rightarrow \mathit{Stop}$$
$$\mathit{traces}\,[\![\, \mathit{Want}_T(\{t_1, t_2\})\,]\!] =$$
$$\{\langle\rangle, \langle \mathit{finish}.t_1\rangle, \langle \mathit{finish}.t_2\rangle, \langle \mathit{finish}.t_1, \mathit{finish}.t_2\rangle, \langle \mathit{finish}.t_2, \mathit{finish}.t_1\rangle\}$$

If the transformation graph can generate all of these datatypes, the *GraphOutputs* process will output exactly one *finish* message for each. Further, since the *finish* messages are not coupled to the order in which the atomic transformations are executed, *GraphOutputs* will be able to output these *finish* messages in any order. Thus, the traces of Want_T will be a subset of the traces of *GraphOutputs*. (In fact, because neither process has any other visible events, they will be traces-equivalent.) On the other hand, if the graph *cannot* generate each desired datatype, then the *GraphOutput* process will not have any trace containing every *finish* event. Since Want_T does contain such a trace, the traces of Want_T will *not* be a subset of the traces of *GraphOutputs*. Thus, the refinement $\mathit{GraphOutputs} \sqsubseteq_T \mathit{Want}_T$ will succeed iff there is a valid solution.

Unfortunately, while this correctly tells us if a compound transformation *exists*, it does not tell us what the transformation *is*. Luckily, we can find this information with only slight modifications. We create a new Want_F process as follows:

$$\mathit{Want}_F(\varnothing) = \mathit{Stop}$$
$$\mathit{Want}_F(\{t\}) = \mathit{Stop}$$
$$\mathit{Want}_F(\mathit{types}) = \sqcap\, t : \mathit{types} \bullet \mathit{finish}!t \rightarrow \mathit{Want}_F(\mathit{types} \setminus \{t\})$$

This differs from Want_T in two respects. First, we use internal choice instead of interleaving to establish each permutation of the *finish* events. Second, for of each these permutations, we only accept *all but one* of the *finish* events, refusing the final one.

With these changes, we can use a *stable failures* refinement instead. If there is a valid compound transformation, the *GraphOutputs* process must allow every *finish* message to occur, in any permutation. The $Want_F$ process, however, only accepts all but one of these events; there is no situation where it will accept every *finish* event. Thus, the stable failures of *GraphOutputs* are *not* a subset of the stable failures of $Want_F$.

If, on the other hand, no compound transformation is possible, then there must be at least one *finish* event that *GraphOutputs* refuses. Further, it will refuse this *finish* event at every point during its execution. $Want_F$ can also refuse this event at any point: either because there are other *finish* events for the internal choice to fall back on, or because it is the final remaining *finish* event, which it always refuses. Thus, the stable failures of *GraphOutputs are* a subset of the stable failures of $Want_F$. The refinement $Want_F \sqsubseteq_F$ *GraphOutputs* will therefore *fail* iff there is a valid solution. (Note that our choice of model is important. The *Graph* process can execute the same transformation repeatedly forever, which causes the *GraphOutputs* process to diverge. By using the stable failures model instead of the failures-divergences model, we ignore these situations.)

Now, when we ask FDR to check this refinement, there are two possible outcomes. If the refinement check succeeds, then we know that there is no valid compound transformation. If it fails, then the compound transformation exists, and FDR will provide a counterexample to the refinement. By examining this counterexample, we will find the sequence of *execute* events that defines the compound transformation solution.

4. Analysis using FDR

In the previous section we presented a declarative description of the polyadic discovery problem using the CSP process algebra. By casting the problem as a suitable refinement test between two processes, we can use the FDR refinement checker as a prototype implementation. In this section, we run this refinement check over many different transformation graphs, of varying shapes and sizes, recording how efficiently FDR can find solutions. Doing so gives us an empirical view of the complexity space of the problem, with the hope of finding regions of inputs for which the discovery algorithm is more efficient than the *NP*-hard worst-case bound. Ideally, these regions will correspond to the kinds of transformation graphs that are more likely to appear in practice, suggesting that there is an algorithmic solution that will be useful in the normal case.

The obvious way to measure the space and time complexity of our prototype would be to record the maximal amount of memory used by FDR, and the amount of wallclock or actual processor time that it takes to perform the refinement check. However, we use a different metric: all measurements are made with respect to the underlying labeled transition system (LTS) that FDR creates for a compiled CSP process. Because of *supercompilation* [Goldsmith 2005], FDR will usually not have to store the process's entire abstract LTS in memory. We measure the space complexity as the size of this smaller supercompiled LTS. The refinement check, however, must be performed on the full abstract LTS, which requires *explicating* the supercompiled LTS into its full form. (The explicated LTS nodes are allocated and deallocated as they are needed, so as to avoid storing the full LTS in memory at once.) We therefore use the number of explicated LTS states visited during the refinement check as a measure of the time complexity.

We measure the space and time complexity in this way because these measure-

ments depend only on the definition of the CSP process. The space complexity metric is fully deterministic, since FDR will always compile a CSP process into the same LTS. The time metric is fully deterministic, as well, since FDR will perform the same search for any particular refinement check. Our measurements, therefore, do not depend on the speed or load of the machine used to perform the refinement check, and are more reproducible.

4.1. Space complexity

Our first experiment is to measure the space complexity of the constructed graph representation. The "before" curves in Figure 3 show the size of the labeled transition system that FDR constructs for the transformation graph processes. Initially, we only consider how the graph size is affected by the number of datatypes in the graph, so we consider graphs containing a varying number of datatypes and no transformations. As the figure shows, the graph size grows very quickly; graphs with more than twenty datatypes took over an hour to compile on a reasonably fast workstation.

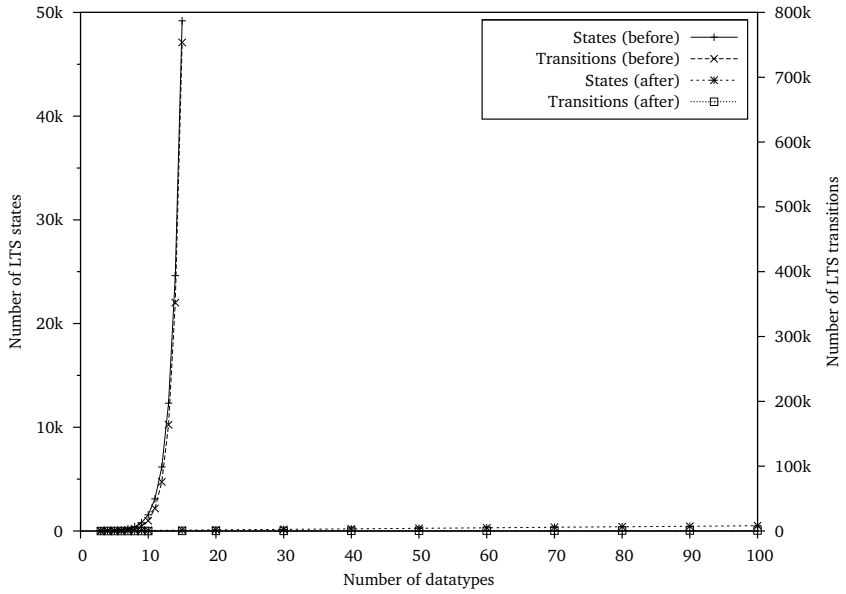

Figure 3. Space required for the transformation graph process

The problem is with the *Finish* and *XformPrereq* processes, specifically with their internal *Have* subprocesses. These subprocesses track the set of available datatypes as a state parameter. Unfortunately, sets require exponential space; since FDR is compiling this subprocess into a low-level operator tree, the *Have* process's LTS also requires exponential space. Luckily, we can modify the *Finish* process as follows:

$$
\begin{array}{l}
Finish = \\
\quad \textsf{let} \\
\qquad DontHave(t) = have!t \rightarrow Have(t) \\
\qquad Have(t) = \\
\qquad\quad (t \in DesiredTypes)\ \&\ finish!t \rightarrow Finished(t) \\
\qquad\quad \Box \\
\qquad\quad have!t \rightarrow Have(t) \\
\qquad Finished(t) = have!t \rightarrow Finished(t)
\end{array}
$$

$$
\begin{array}{l}
\quad \textsf{within} \\
\quad\quad \Vert\!\Vert\ t : \mathit{Datatype} \bullet \mathit{DontHave}(t)
\end{array}
$$

We can make a similar modification to *XformPrereq*:

$$
\begin{array}{l}
\mathit{XformPrereq}(\ (\mathit{id}, \mathit{inputTypes}, \mathit{outputTypes})\ ) = \\
\quad \textsf{let} \\
\quad\quad \alpha(\mathit{DontHave}(t)) = \{\mathit{execute.id}, \mathit{have.t}\} \\
\quad\quad \mathit{DontHave}(t) = \mathit{have}!t \rightarrow \mathit{Have}(t) \\
\quad\quad \mathit{Have}(t) = (\mathit{execute}!\mathit{id} \rightarrow \mathit{Have}(t)) \ \Box\ (\mathit{have}!t \rightarrow \mathit{Have}(t)) \\
\quad \textsf{within} \\
\quad\quad \Vert\ t : \mathit{inputTypes} \bullet \mathit{DontHave}(t)
\end{array}
$$

Here we have redefined the internal subprocesses to only keep track of a single datatype. We then create copies of these internal subprocesses for each of the datatypes, and use a composition operator to combine them. For the *Finish* process, we can use interleaving, since the subprocess alphabets are disjoint. In the *XformPrereq* process, on the other hand, the subprocesses for each input datatype must synchronize on the *execute* event, since all of the inputs must be available before the transformation can proceed. We must therefore use alphabetized parallel for the composition. FDR will compile these subprocesses into low-level operator trees; however, since they no longer maintain exponential state, these trees will be small. The composition of these smaller processes is far more efficient than the original exponential LTS; the "after" curves in Figure 3 show the same space measurements for a graph constructed with the modified subprocesses. With this modification, we are easily able to represent graphs with hundreds of datatypes.

Next we show how the size of the graph process is affected by the number and arrangement of transformations in the graph. For this experiment, we fix the number of datatypes in the graph, and examine four situations. First, as a control, we examine the graph with no transformations. Second, we introduce a single directed cycle of transformations that encompasses all of the datatypes in the graph. Third, we consider a graph with two directed cycles, pointing in opposite directions. Finally, we consider the fully-connected graph, where a transformation directly connects every possible pair of datatypes.

Figure 4 shows how the size of the LTS depends on the number of datatypes and transformations for each style of graph. Part of the overall growth comes from the datatypes, and part comes from the transformations. The contour lines show that the resulting surface is planar, yielding an $O(D + T)$ overall size for a graph's LTS. The XY plane represents how the number of transformations depends on the number of datatypes for a particular style of graph; projecting a particular graph's curve up onto the growth plane then yields a single growth curve for that style of transformation graph.

4.2. Time complexity

Next we examine the time complexity of the algorithm. We again look at four different "shapes" of transformation graph, shown in Figure 5. In all cases, we are seeking a transformation between the source datatype *S* and the destination datatype *D*. The shapes differ in the number of additional datatypes in the graph, and in how the datatypes are

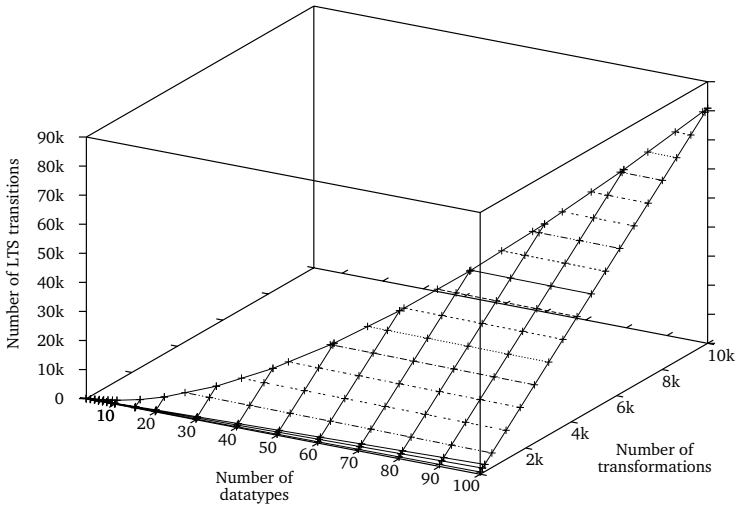

Figure 4. Relationship between datatypes, transformations, and LTS size

connected. In part (a), we have a single sequence of datatypes A_1 through A_n, with a single path through the graph from S to D. In part (b), we have the same sequence of datatypes A_1 through A_n, but in this case, they are not needed to transform from S to D. In part (c), we have two sequences of datatypes, A_1 through A_n and B_1 through B_n, between S and D. Either one can be used as a valid transformation path. Finally, in part (d), we again have two sequences of datatypes between S and D, but we introduce crosslinks as well, allowing the algorithm to jump from the A datatypes to the B datatypes at any point in the sequence. In this graph, there are $n + 2$ valid transformations between S and D.

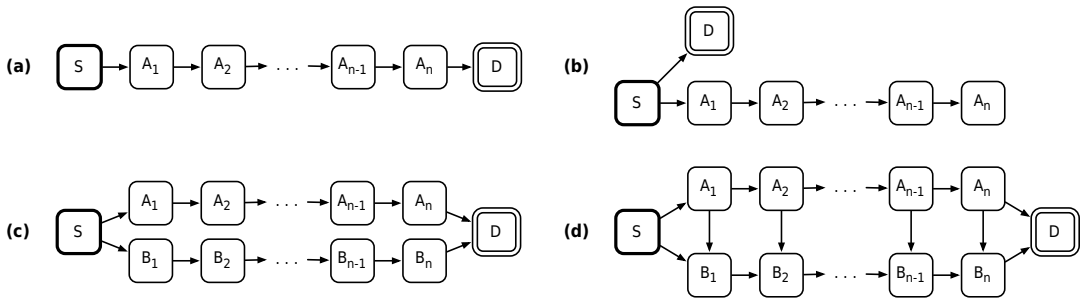

Figure 5. Transformation graph shapes used for timing analysis

The results of this analysis are shown in Figure 6. Several important conclusions can be drawn from this data. In most cases, the number of LTS states and transitions that must be examined during the discovery algorithm is much greater than the number needed to represent the graph itself. This implies that with our more efficient subprocesses, FDR is not initially instantiating the entire structure of the graph; rather, the graph process is encoding a recipe for dynamically instantiating the graph as needed. This corresponds with our understanding of FDR's use of supercompilation to distinguish between low- and high-level operator trees.

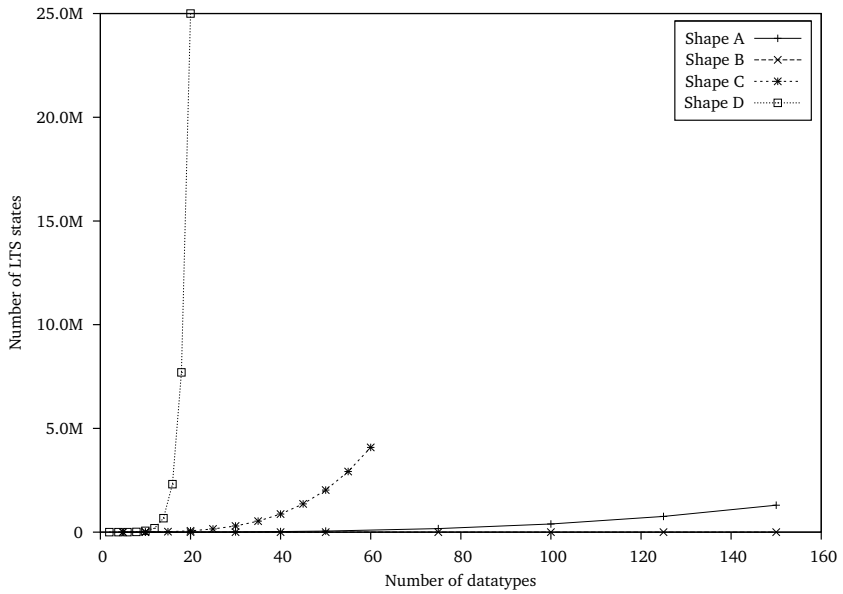

Figure 6. Refinement running time for all shapes

Next, we can see that the execution time for the discovery algorithm is almost entirely dependent on the number of transformation paths that must be checked. This is most apparent in shape B, whose curve disappears into the X axis. Regardless of the number of datatypes in the shape B graph, there is a constant size transformation solution. Once the discovery algorithm finds this path, no more processing is required.

Figure 6 also shows the relationship between the complexity curves of shapes A, C, and D. Shape C seems to be a simple modification to the graph from shape A, only adding a single additional possible transformation path. However, since FDR is performing the equivalent of a breadth-first search, it will try to make progress down both paths simultaneously. Only after reaching the end of both intermediary sequences will FDR discover that only one was needed to reach the destination. Worse, it must consider every interleaving of the transformations in the two paths while advancing through the graph. Shape D exacerbates this problem by introducing crosslinking edges. Now, instead of having to consider all possible permutations of two transformation paths, FDR must consider the permutations of $n + 2$ paths. The exponential growth is much more pronounced; whereas with shape A we were able to consider graphs with 150 datatypes in a reasonable amount of time, shape D quickly becomes intractable after only twenty datatypes.

Finally, it is important to point out we have not eliminated the exponential growth curve of the algorithm's running time; we have only made it less steep. This might seem to be a discouraging result at first, but it is in fact still useful in practice. As we will discuss in the next section, real-world transformation graphs tend not to contain a large number of datatypes and transformations, so any improvement in the efficiency of the discovery algorithm for smaller graphs will be helpful.

5. Interpreting the results

Having expressed the polyadic discovery problem as a CSP process, and analyzed the complexity space of this process, we can now interpret the measurements that we ob-

tained. First, we examine the kinds of transformation graphs that were more efficient, identifying the features of those graphs that led to the efficiency gains, and showing why "real-life" transformation graphs will tend to have those features. Then, we show how this information can be used to construct an algorithmic solution.

5.1. Causes of the efficiency gains

According to our analysis, the space complexity for the discovery algorithm is fairly static, determined by the number of datatypes and transformations in the graph. This implies that reducing the number of datatypes in a graph can be an effective means of improving efficiency. In practice, this strategy should prove useful, since large transformation graphs tend to be easily separated into connected components. Intuitively, this is because the datatypes in the graph will tend to form "clumps", where a datatype can be transformed into anything in its clump, but not into anything outside of it. By treating these connected components as separate transformation graphs, we reduce the number of datatypes and the space required to represent the graph.

The time complexity, on the other hand, depends much more on the "shape" of the graph. As suspected, certain input graphs provide much more efficient executions of the discovery algorithm. The major determining factor is the number of possible transformation paths that must be checked. The time required by FDR grew dramatically as edges were added to the graph, especially when those edges added new transformation paths without making new transformations reachable. This yields a portion of the graph where several different possible sequences of transformations must be considered. Each of these sequences will eventually yield the same set of available datatypes, but will require different intermediary sets to get there. In practice, transformation graphs will usually avoid this inefficiency: the clumps of datatypes in a graph will not be highly interconnected, since the entire reason for using this graph-based approach is to limit the need to write direct transformations between datatypes.

A similar factor affecting the algorithm's time complexity is whether the compound transformation that we are seeking actually exists. FDR is able to find a solution much faster than it is able to prove that no solution exists. Intuitively, this makes sense; once a solution is found, FDR does not need to consider any of the remaining possibilities and can stop processing. If there is no solution, FDR must check every state of the LTS to prove this. In practice, a program or user will use this discovery algorithm because they know (or can reasonably assume) that the desired compound transformation exists. For real-world use cases, therefore, the time complexity will tend to be more efficient.

5.2. Developing an algorithm

With the insights gained by the CSP description of the problem and the efficiency analysis of using FDR as a prototype implementation, we can now construct an algorithmic solution to the problem. In our initial formulation, our CSP processes kept track of which datatypes were available at any given time. We can do the same using a graph, where the nodes represent sets of datatypes. Each atomic transformation then yields several edges in the graph. For each node that contains all of a transformation's inputs, an edge is added from that node. The edge's destination is the union of the datatypes that were available previously (the source node) and the datatypes created by the transformation (the trans-

formation's output set). This *set graph* representation is shown in Figure 7(a) for the example transformation graph from Figure 2.

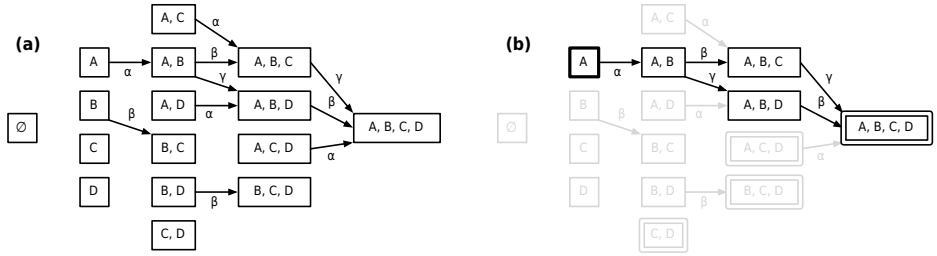

Figure 7. A set-based graph representation

Using this model, compound transformations are once again represented by paths. Figure 7(b) shows the possible solutions that result if we are given an instance of datatype A and want to generate instances of datatypes C and D. There are two solutions, since β and γ can be executed in either order. Since solutions are represented by paths, we can use a shortest path algorithm to discover compound transformations.

However, we now have the same problem as with the initial CSP implementation: the space requirement for this representation is exponential in the number of datatypes. The next step in our analysis was to modify the CSP process to store a recipe for lazily deriving a transformation graph's LTS, rather than storing it in memory in its entirety.

This strategy of lazy evaluation can be easily added to the set-based graph algorithm, since a large fraction of nodes are not reachable from the $\{A\}$ source node. By only instantiating the nodes in the graph as they are encountered during the pathfinding algorithm, and by stopping the processing once a shortest path has been discovered, we will only instantiate the subset of nodes that are actually reachable. Further, as we have shown, these reachable subsets will tend to be small for the transformation graphs that will appear more often in practice. Thus, while we cannot improve the problem's worst-case intractability, we have been able to empirically derive an algorithm that allows "real-world" transformations to be discovered efficiently and effectively.

6. Discussion

In this paper, we have described a transformation discovery problem that is provably intractable in the worst case. However, many problems are intractable in the worst case only because of pathological examples that do not arise in practice, and are more efficient in the "normal" case. We therefore formulated a declarative CSP description of the problem, and used FDR as a prototype implementation. This allowed us to identify classes of inputs for which solutions could be found more efficiently; using this information, we were then able to develop an algorithmic solution that is useful for real-world inputs.

One limitation of this technique is that we must choose an appropriate sampling of inputs if we want a true view of the problem's complexity space. More importantly, even if we choose an appropriate suite of test inputs, there is no guarantee that FDR will find all of the efficient solutions that are possible. When FDR finds a compound transformation inefficiently, for instance, this does not mean that this input graph has no efficient solution; instead, it might be that FDR's refinement checking strategies cannot

reproduce the necessary optimizations. In general, negative results are not indicative. Positive results, on the other hand, represent real optimizations that can be exploited, though even these results might not be fully optimal.

Further work in this area can proceed along three lines of enquiry. First, if we want more confidence in our view of the problem's complexity space, we could create several prototypes, each using a different underlying declarative language, hoping that each efficient class of inputs would be found by at least one of them. SAT solvers, in particular, would be a good choice for an additional prototype; being the earliest and most visible *NP*-hard problem [Cook 1971], Boolean satisfiability has inspired research into many sophisticated optimization techniques [Gu et al. 1997].

Second, there has been a lot of research into compression and optimization techniques for CSP processes. The supercompilation approach described in [Goldsmith 2005] is integral to the lazy evaluation strategy that we have already exploited. The hierarchical compression functions described in [Roscoe et al. 1995] seem promising, as well. It would be fruitful to see if any of these CSP compressions could be used to obtain further optimizations.

Finally, this technique could be similarly used for any algorithm or problem that can expressed as a refinement of CSP processes — by finding the inputs that are solved more efficiently by FDR, and searching for common features of those inputs. This would then hopefully provide insights into how the algorithm could be made more efficient for those cases. One could verify this by applying this technique to several well-known *NP*-hard problems, seeing if it can reproduce existing results and lead to new insights.

Acknowledgments

Doug Creager's work is funded by the Software Engineering Programme of the Oxford University Computing Laboratory. The authors would like to thank Phil Armstrong for his advice on automating the FDR refinement checks in this paper; Jeremy Gibbons and Ed Smith for their suggestions on SAT solvers and other model-checking techniques; and our anonymous referees for providing valuable on the manuscript of this paper.

References

Ausiello, G., D'Atri, A., and Saccà, D. (1983). Graph algorithms for functional dependency manipulation. *Journal of the ACM*, 30:752–766.

Ausiello, G., Italiano, G. F., and Nanni, U. (1992). Optimal traversal of directed hypergraphs. Technical Report TR-92-073, ICSI, Berkeley, CA.

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90.

Berge, C. (1973). *Graphs and hypergraphs*, volume 6 of *North-Holland Mathematical Library*. Elsevier, Amsterdam.

Berge, C. (1989). *Hypergraphs: Combinatorics of finite sets*, volume 45 of *North-Holland Mathematical Library*. Elsevier, Amsterdam.

Breese, J. S., Heckerman, D., and Kadie, C. (1998). Empirical analysis of predictive algorithms for collaborative filtering. Technical Report MSR-TR-98-12, Microsoft Research.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *3rd ACM Symp. on the Theory of Computing*, pages 151–158, New York. ACM Press.

Creager, D. A. and Simpson, A. C. (2006). A fully generic, graph-based approach to data transformation discovery. In *Graph Computation Models (GCM06)*.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Esparza, J. (1998). Decidability and complexity of Petri net problems — an introduction. In *Lectures on Petri Nets I: Basic Models*, LNCS 1491, pages 374–428. Springer-Verlag, Berlin.

Ford, Jr., L. R. and Fulkerson, D. R. (1962). *Flows in Networks*. Princeton University Press.

Goldsmith, M. (2005). Operational semantics for fun and profit. In Abdallah, A. E., Jones, C. B., and Sanders, J. W., editors, *Communicating Sequential Processes: The First 25 Years*, LNCS 3525, pages 265–274. Springer-Verlag.

Gu, J., Purdom, P. W., Franco, J., and Wah, B. W. (1997). *Algorithms for the satisfiability (SAT) problem: A survey*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Hunt, J. J., Vo, K.-P., and Tichy, W. F. (1998). Delta algorithms: an empirical analysis. *ACM Trans. on Software Engineering Methodologies*, 7(2):192–214.

Italiano, G. F. and Nanni, U. (1989). On-line maintenance of minimal directed hypergraphs. In *3rd Italian Conf. on Theoretical Computer Science*, pages 335–349. World Scientific Co.

Jones, D. W. (1986). An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311.

Petri, C. A. (1962). *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn.

Petri, C. A. (1963). Fundamentals of a theory of asynchronous information flow. In *Int'l Fed. for Information Processing Congress (IFIP 62)*, pages 386–390.

Roscoe, A. W. (1994). Model-checking CSP. In Roscoe, A. W., editor, *A classical mind: Essays in honour of C. A. R. Hoare*, pages 353–378. Prentice-Hall.

Roscoe, A. W. (1998). *The theory and practice of concurrency*. Prentice-Hall.

Roscoe, A. W., Gardiner, P. H. B., Goldsmith, M. H., Hulance, J. R., Jackson, D. M., and Scattergood, J. B. (1995). Hierarchical compression for model-checking CSP: How to check 10^{20} dining philosophers for deadlock. In *1st Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1019, pages 133–152, Berlin. Springer-Verlag.

Scattergood, J. B. (1998). *The semantics and implementation of Machine-Readable CSP*. D.Phil. dissertation, Oxford University.