On the Computational Complexity of Designing and Reconfiguring Component-based Software Systems

Todd Wareham [*]
Department of Computer Science
Memorial University of Newfoundland
St. John's, NL Canada
harold@mun.ca

Marieke Sweers
Radboud University Nijmegen
Donders Institute for Brain, Cognition, and Behaviour
Nijmegen, The Netherlands
marieke.sweers@gmail.com

ABSTRACT

Though Component-Based Development (CBD) is a popular approach to mitigating the costs of creating software systems, it is not clear to what extent CBD is preferable to other approaches to software engineering or to what extent the core component selection and adaptation activities of CBD can be implemented to operate without human intervention in an efficient and reliable manner. In this paper, we use computational complexity analysis to compare the computational characteristics of software system design and reconfiguration by *de novo* design, component selection, and component selection with adaptation. Our results show that none of these approaches can be implemented to operate both efficiently and reliably either in general or relative to a surprisingly large number of restrictions on software system, component, and component library structure. We also give the first restrictions under which all of these approaches can be implemented to operate both efficiently and reliably.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Resuable Software—*Reuse models*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical algorithms and problems—*computations on discrete structures*

General Terms

Algorithms, Design, Measurement, Performance, Theory

Keywords

component-based development, parameterized computational complexity

[*]Corresponding Author

1. INTRODUCTION

Component-based development (CBD) is a popular approach to mitigating the monetary and time costs of creating and maintaining software systems [1, 3, 12]. In CBD, previously-developed software modules called components are stored in libraries and connected together (possibly with some adaptation of the component code and architecture) as needed to generate new software systems from given requirements. Over the last 20 years, a number of technologies have been implemented to assist human programmers in creating component-based systems, *e.g.*, CORBA, CCM, EJB, COM+, and much effort has been put into automating the key CBD activities of component selection and adaptation.

There are two ongoing issues of great importance in the CBD community: (1) the circumstances (if any) under which the cost of selecting and adapting components is less than the cost of developing software systems from scratch, *i.e.*, *de novo* [10] and (2) the need for CBD to operate fully automatically and efficiently to accommodate ubiquitous and autonomic computing systems which need to create and reconfigure software without or with minimal human intervention at runtime [7, 13]. To address these issues, it would be most useful to know if efficient and reliable methods are available for selecting and adapting components in CBD and, if so, under what circumstances these methods outperform other approaches to software engineering.

In this paper, we present results addressing both of these questions. First, using computational complexity analysis [8], we show that both creating and reconfiguring software systems either by *de novo* design, component selection, or component selection with adaptation are all NP-hard and thus intractable in general. Second, using parameterized complexity analysis [5], we give restrictions under which all of these activities are tractable and prove that surprisingly few restrictions on either software system structure or the allowable degree of adaptation render these activities tractable. Though these results are derived relative to basic models of software systems and components, we show that they also apply to more realistic models.

The remainder of this paper is organized as follows. In Section 2, we present our software system, component, and component library models and formalize the problems of *de novo* and component-based software system design and reconfiguration relative to these models. Section 3 demonstrates the intractability of all of these problems in general. Section 4 describes a methodology for identifying conditions for tractability, which is then applied in Section 5 to identify such conditions for our problems. In order to accommodate conference page limits and focus in the main text on the implications of our results for CBD, proofs of selected results are given in Appendix A. Finally, our conclusions and directions for future work are given in Section 6.

1.1 Related Work

Computational complexity analyses of the component selection problem have been done previously [2, 16, 17]. However, the formalizations used in these analyses focused on the satisfaction of sets of desired system objectives, and did not include specifications of software system and component structure that are detailed enough to investigate the conditions under which restrictions on these structures would make component selection tractable. They also did not address the issue of component adaptation.

2. FORMALIZING COMPONENT-BASED DESIGN AND RECONFIGURATION

Our goal in this paper is to assess whether or not component-based development has advantages over conventional software engineering relative to different activities in the software life cycle. This assessment can be stated most simply along two dimensions:

1. *Software design mode*: creating software using only components drawn from component libraries versus creating software by adapting components drawn from component libraries or creating software *de novo* that is organized in a specified component-like fashion.
2. *Software lifecycle activity*: designing the initial version of a software system relative to a set of requirements versus reconfiguring an existing system to accommodate one or more changes to the requirements.

The possibilities implicit in these dimensions result in the following six informal computational problems:

Software Design
Input: Software requirements R, software-structure specification X
Output: A software system S whose structure is consistent with X and whose operation satisfies R.

Software Design from Adapted Components
Input: Software requirements R, a set of software-component libraries $\mathcal{C} = \{C_1, C_2, \ldots, C_{|C|}\}$.
Output: A software system S whose structure consists of connected components derived from components drawn from the libraries in $\mathcal{C}$ and whose operation satisfies R.

Software Design from Components
Input: Software requirements R, a set of software-component libraries $\mathcal{C} = \{C_1, C_2, \ldots, C_{|C|}\}$.
Output: A software system S whose structure consists of connected components drawn from the libraries in $\mathcal{C}$ and whose operation satisfies R.

Software Reconfiguration
Input: A software system S whose structure is consistent with specification X and whose operation satisfies requirements R, new software requirements R_{new}.
Output: A software system S' derived from S whose structure is consistent with X and whose operation satisfies $R \cup R_{new}$.

Software Reconfiguration from
Adapted Components
Input: A software system S whose structure consists of connected components from a set of software-component libraries $\mathcal{C} = \{C_1, C_2, \ldots, C_{|C|}\}$ and whose operation satisfies requirements R, new software requirements R_{new}.
Output: A software system S' derived from S whose structure consists of connected components derived from components drawn from the libraries in $\mathcal{C}$ and whose operation satisfies $R \cup R_{new}$.

Software Reconfiguration from Components
Input: A software system S whose structure consists of connected components from a set of software-component libraries $\mathcal{C} = \{C_1, C_2, \ldots, C_{|C|}\}$ and whose operation satisfies requirements R, new software requirements R_{new}.
Output: A software system S' derived from S whose structure consists of connected components drawn from the libraries in $\mathcal{C}$ and whose operation satisfies $R \cup R_{new}$.

To assess the computational difficulty of and (if necessary) explore algorithmic options for efficiently solving these problems, we need formalizations of all of the entities comprising these problems. There is a vast existing literature on various types of software requirements, system-structures, structure specifications, and components and component libraries. We want to avoid introducing spurious computational difficulty due to powerful formalisms, as this would complicate the interpretation our results. Hence, we shall choose the simplest such formalizations[1] (additional advantages gained by such a parsimonious approach are given in Section 5.2):

- *Software Requirements*: The requirements will be a set $R = \{r_1, r_2, \ldots r_{|R|}\}$ of situation-action pairs where each pair $r_j(s_j, a_j)$ consists of a situation s_j defined by a particular set of truth-values $s_j = \langle v(i_1), v(i_2), \ldots v(i_{|I|})\rangle$, $v(i_k) \in \{True, False\}$, relative to each of the Boolean variables i_k in set $I = \{i_1, i_2, \ldots, i_{|I|}\}$ and an action a_j from set $A = \{a_1, a_2, \ldots a_{|A|}\}$.
- *Software system-structure*: We will here consider the simplest possible complex structure consisting of a multiple `IF-THEN-ELSE` statement block (a **selector**) whose branches are in turn other `IF-THEN-ELSE` statement blocks (**procedures**) whose branches execute actions from A. Each selector and procedure `IF-THEN` condition is a Boolean formula that is either a variable from I *e.g.*, i_j, or its negation, *e.g.*, $\neg i_j$. Each selector and

[1]These formalizations are based on those developed in [15, 19] to analyze the computational complexity of the adaptive toolbox theory of human decision-making [9].

req.	i_1	i_2	i_3	i_4	i_5	action
r_1	T	T	T	T	T	a_2
r_2	T	F	F	F	T	a_1
r_3	F	F	F	F	F	a_2
r_4	F	F	F	F	F	a_2
r_5	T	T	T	F	T	a_3

(a)

```
procedure p1:                procedure p2
  if $i_4$ then $a_2$                 if $\neg i_2$ then $a_1$
  else if $\neg i_3$ then $a_1$         else if $\neg i_4$ then $a_2$
  else if $i_5$ then $a_3$          else $a_3$
  else $a_1$

procedure p3:                procedure p4:
  if $i_4$ then $a_2$                 $a_2$
  else $a_2$
```

(b)

```
selector s1:               selector s2:
  if $i_1$ then ???              if * then ???
  else if $i_5$ then ???
  else ???
```

(c)

```
system S1:                    system S2:
  if $i_1$ then call p1             if * then call p2
  else if $i_5$ then call p3
  else call p4
```

(d)

Figure 1: Example Requirements, Procedures, Selectors, and Software Systems. (a) Software requirements $R = \{r_1, r_2, r_3, r_4, r_5\}$ defined on situation-variables $I = \{i_1, i_2, i_3, i_4, i_5\}$ and action-set $A = \{a_1, a_2, a_3\}$. (b) Four procedures. (c) Two selectors as they would appear in a selector-component library with blank procedure calls. (d) Two software systems created by instantiating the procedure-calls in the selectors from (c) with procedures from (b).

procedure with one or more situation-variable conditions terminates in a final `ELSE` statement. In addition, the selector may consist of single special statement `IF * THEN` which evaluates to *True* in all cases (the default selector) and a procedure may consist of a single executed action; such selectors and procedures have no associated conditions.

- *Software components and component libraries*: Two-level software systems of the form above consist of two types of components, selectors and procedures, which shall be stored in libraries L_{sel} and L_{prc}, respectively.

- *Software structure specification*: The basic characteristics of two-level software systems of the form above are the situation-variable and action sets on which they are based (I and A), the maximum number of conditions allowed in the selector ($|sel|$) and the maximum number of conditions allowed in any procedure ($|prc|$).

Examples of these entities are given in Figure 1.

This allows us to formalize various actions and properties in the problems given above as follows:

- A software system S is consistent with a structure-specification $X = \langle I, A, |sel|, |prc|\rangle$ if it has the two-level structure described above where `IF-THEN` conditions are (un)negated members of I, all procedure-executed actions are drawn from A, the number of conditions in the selector is at most $|sel|$, and the number of conditions in each procedure is at most $|prc|$. For example, both software-systems in Figure 1(d) are consistent with $X = \{\{i_1, i_2, i_3, i_4, i_5\}, \{a_1, a_2, a_3\}, 2, 3\}$.

- A software system S is constructed from components drawn from $\mathcal{C} = \{C_{sel}, C_{prc}\}$ if the selector-component is from C_{sel} and each procedure-component is from C_{prc}. Note that a member of C_{prc} may appear zero, one, or more times in S. For example, both software systems in Figure 1(d) are constructed from $\mathcal{C} = \{\{\texttt{s1}, \texttt{s2}\}, \{\texttt{p1}, \texttt{p2}, \texttt{p3}, \texttt{p4}\}\}$.

- The operation of a software system S satisfies requirements R if for each situation-action pair (s, a) in R, the execution of S relative to the truth-settings in s results in the execution of a. For example, software system `S1` in Figure 1(d) satisfies the requirements in Figure 1(a) but software system `S2` does not (because it produces different outputs (a_3, a_1, a_1, and a_2 respectively) for requirements r_1, r_3, r_4, and r_5).

We can now state the following formal problems:

SOFTWARE DESIGN (SDes-Spec)
Input: Software-structure specification $X = \langle I, A, |sel|, |prc|\rangle$, software requirements R based on sets I and A.
Output: A software system S whose structure is consistent with X and whose operation satisfies R, if such an S exists, and special symbol $\perp$ otherwise.

SOFTWARE DESIGN FROM
ADAPTED COMPONENTS (SDes-CompA)
Input: Software requirements R based on sets I and A, a set of software-component libraries $\mathcal{C} = \{C_{sel}, C_{prc}\}$, positive integers $d, c_c \geq 0$.
Output: A software system S whose structure consist of connected components derived by at most c_c changes to at most d components drawn from the libraries in $\mathcal{C}$ and whose operation satisfies R, if such an S exists, and special symbol $\perp$ otherwise.

SOFTWARE DESIGN FROM COMPONENTS (SDes-Comp)
Input: Software requirements R based on sets I and A, a set of software-component libraries $\mathcal{C} = \{C_{sel}, C_{prc}\}$, a positive integer $d \geq 0$.
Output: A software system S whose structure consist of connected components based on at most d components drawn from the libraries in $\mathcal{C}$ and whose operation satisfies R, if such an S exists, and special symbol $\perp$ otherwise.

SOFTWARE RECONFIGURATION (SRec-Spec)
Input: A software system S whose structure is consistent with specification $X = \langle I, A, |sel|, |prc|\rangle$ and whose operation satisfies requirements R, a set R_{new} of new situation-action pairs based on I and A, a positive integer $c_c \geq 0$.

Output: A software system S' derived by at most c_c changes to S whose structure is consistent with X and whose operation satisfies $R \cup R_{new}$, if such an S' exists, and special symbol $\perp$ otherwise.

Software Reconfiguration from
Adapted Components (SRec-CompA)
Input: A software system S whose structure consists of connected components from a set of software-component libraries $\mathcal{C} = \{C_{sel}, C_{prc}\}$ and whose operation satisfies requirements R, a set R_{new} of new situation-action pairs based on I and A, positive integers $c_l, c_c, d \geq 0$.
Output: A software system S' derived from S by at most c_l component-changes relative to the libraries in $\mathcal{C}$ and c_c code-changes whose structure consist of at most d types of connected components and whose operation satisfies $R \cup R_{new}$, if such an S' exists, and special symbol $\perp$ otherwise.

Software Reconfiguration from
Components (SRec-Comp)
Input: A software system S whose structure consists of connected components from a set of software-component libraries $\mathcal{C} = \{C_{sel}, C_{prc}\}$ and whose operation satisfies requirements R, a set R_{new} of new situation-action pairs based on I and A, positive integers $c_l, d \geq 0$.
Output: A software system S' derived from S by at most c_l component-changes relative to the libraries in $\mathcal{C}$ whose structure consists of at most d types of connected components and whose operation satisfies $R \cup R_{new}$, if such an S' exists, and special symbol $\perp$ otherwise.

We have included parameter d in all component-based problems to allow control over the number of retrievals from libraries. The two types of adaptation and reconfiguration changes are: (1) changes to the system code: changing the condition in or action executed by any `IF-THEN-ELSE` statement (c_c) and (2) changes to the system component-set: using a different selector-component from C_{sel} or changing any used procedure-component to another from C_{prc} (c_l). Note in the case of a selector-change, both the original and new selector must have the same number of `IF-THEN` statements and the procedures called by the original selector are mapped to the corresponding positions in the new selector.

3. COMPONENT-BASED DESIGN AND RECONFIGURATION ARE INTRACTABLE

Following general practice in Computer Science [8], we define tractability as being solvable in the worst case in time polynomially bounded in the input size. We show that a problem is not polynomial-time solvable, *i.e.*, not in the class P of polynomial-time solvable problems, by proving it to be at least as difficult as the hardest problems in problem-class NP (see [8] for details).

Result A: SDes-Spec, SDes-CompA, SDes-Comp, SRec-Spec, SRec-CompA, and SRec-Comp are NP-hard.

Modulo the conjecture $P \neq NP$ which is widely believed to be true [6], the above shows that even the basic versions of component-based design and reconfiguration considered here are not solvable in polynomial time. This NP-hardness hold for problems SRec-Spec and SRec-CompA when the new requirements consist of a single situation-action pair.

Two frequently-adopted responses to intractability are to consider poly-time algorithms which generate solutions that (1) approximate within provable bounds the wanted solutions, *e.g.*, software systems whose structure are close to the values specified in X or which satisfy a large proportion of the requirements in R [14], or (2) are of acceptable quality most of the time, *e.g.*, genetic or simulated annealing algorithms [11]. In the remainder of this paper, we will consider a third option described in more detail in the next section.

4. A METHOD FOR IDENTIFYING TRACTABILITY CONDITIONS

A computational problem that is intractable for unrestricted inputs may yet be tractable for non-trivial restrictions on the input. This insight is based on the observation that some NP-hard problems can be solved by algorithms whose running time is polynomial in the overall input size and non-polynomial only in some aspects of the input called **parameters**. The following states this idea more formally.

Definition 1. Let Π be a problem with parameters $k_1, k_2, \ldots$. Then Π is said to be **fixed-parameter (fp-) tractable** for parameter-set $K = \{k_1, k_2, ...\}$ if there exists at least one algorithm that solves Π for any input of size n in time $f(k_1, k_2, ...)n^c$, where $f(\cdot)$ is an arbitrary function and c is a constant. If no such algorithm exists then Π is said to be **fixed-parameter (fp-) intractable** for parameter-set K.

In other words, a problem Π is fp-tractable for a parameter-set K if all superpolynomial-time complexity inherent in solving Π can be confined to the parameters in K.

There are many techniques for designing fp-tractable algorithms [4, 5], and fp-intractability is established in a manner analogous to classical polynomial-time intractability by proving a parameterized problem is at least as difficult as the hardest problems in one of the problem-classes in the W-hierarchy $\{W[1], W[2], ...\}$ (see [5] for details). Additional results are typically implied by any given result courtesy of the following lemmas:

Lemma 1. *[20, Lemma 2.1.30] If problem Π is fp-tractable relative to parameter-set K then Π is fp-tractable for any parameter-set K' such that $K \subset K'$.*

Lemma 2. *[20, Lemma 2.1.31] If problem Π is fp-intractable relative to parameter-set K then Π is fp-intractable for any parameter-set K' such that $K' \subset K$.*

It follows from the definition of fp-tractability that if an intractable problem Π is fp-tractable for parameter-set K, then Π can be efficiently solved even for large inputs, provided only that the values of all parameters in K are relatively small. This strategy has been successfully applied to many intractable problems (see [5, 18] and references). In the next section we show how this strategy may be used to render component-based design and reconfiguration tractable.

Table 1: Parameters for Component-based Design and Reconfiguration Problems

Parameter	Description	Applicability
$\|I\|$	# situation-variables	All
$\|A\|$	# possible actions	All
$\|sel\|$	Max # selector-conditions	All
$\|prc\|$	Max # procedure-conditions	All
d	Max # component-types in software system	*-CompA, *-Comp
$\|L_{sel}\|$	# selector-components in selector-library	*-CompA, *-Comp
$\|L_{prc}\|$	# procedure-components in procedure-library	*-CompA, *-Comp
$\|R_{new}\|$	# new requirements	SRec-*
c_c	Max # code-changes allowed	*-CompA, SRec-Spec
c_l	Max # component-changes allowed	SRec-CompA, SRec-Comp

5. WHAT MAKES COMPONENT-BASED DESIGN AND RECONFIGURATION TRACTABLE?

Our component-based software design and reconfiguration problems have a number of parameters whose restriction could render these problems tractable in the sense defined in Section 4. An overview of the parameters that we considered in our fp-tractability analysis is given in Table 1. These parameters can be divided into three groups:

1. Restrictions on system and component structure ($|I|$, $|A|$, $|sel|$, $|prc|$, d);
2. Restrictions on component libraries structure ($|L_{sel}|$, $|L_{prc}|$); and
3. restrictions on reconfigurability ($|R_{new}|, c_c$, c_l).

We will assess the fixed-parameter tractability of our problems relative to the parameters in Table 1 (Section 5.1), show how these results apply in more general settings (Section 5.2), and discuss the implications of these results for component-based development (Section 5.3).

5.1 Results

Our parameterized intractability results are as follows:

Result B: SDes-Spec is $W[2]$-hard for $\{|A|, |sel|, |prc|\}$ when $|A| = 2$ and $|sel| = 0$.

Result C: SDes-CompA is $W[2]$-hard for $\{|A|, |sel|, |prc|, d, |L_{sel}|, |L_{prc}|, c_c\}$ when $|A| = d = 2$, $|sel| = 0$, $|L_{sel}| = |L_{prc}| = 1$.

Result D: SDes-Comp is $W[2]$-hard for $\{|A|, d, |L_{sel}|\}$ when $|A| = 2$ and $|L_{sel}| = 1$.

Result E: SRec-Spec is $W[2]$-hard for $\{|A|, |sel|, |prc|, |R_{new}|, c_c\}$ when $|A| = 3$, $|sel| = 0$, and $|R_{new}| = 1$.

Result F: SRec-CompA is $W[2]$-hard for $\{|A|, |sel|, |prc|, d, |L_{sel}, |L_{prc}|, |R_{new}|, c_l, c_c\}$ when $|A| = 3$, $|R_{new}| = |L_{sel}| = |L_{prc}| = 1$, $d = 2$, and $|sel| = c_l = 0$.

Result G: SRec-Comp is $W[2]$-hard for $\{|A|, d, |L_{sel}|\}$ when $|A| = 3$ and $|L_{sel}| = 1$.

At present, we have the following fp-tractability results:

Result H: SDes-Spec, SDes-CompA, SDes-Comp, SRec-Spec, SRec-CompA, and SRec-Comp are fixed-parameter tractable for $\{|I|\}$.

Result I: SDes-Comp and SRec-Comp are fixed-parameter tractable for $\{|sel|, |L_{prc}|\}$.

Note that Results B, C, E, F, and H in combination with Lemmas 1 and 2 completely characterize the parameterized complexity of problems SDes-Spec, SDes-CompA, SRec-Spec, and SRec-CompA relative to the parameters in Table 1.

5.2 Generality of Results

Our intractability results, though defined relative to basic models of software requirements, software systems, components, and component libraries, have a broad applicability. Observe that the models for which these results hold are in fact restricted versions of more realistic alternatives, *e.g.*,

- software requirements that explicitly list all situations to which software should respond are a special case of more complex requirements which are based on compact specifications of software behaviour such as finite-state automata or statecharts and/or incorporate other required properties of software systems such as degree of reliability and response time;
- two-level selector / procedure software systems that act as simple functions are a special case of more complex persistent software systems whose components invoke each other in more complex manners;
- components consisting of a single condition-statement block procedure without input parameters or return values and which have no dependencies on other components are a special case of more complex components consisting of multiple data types and complex procedures which have dependencies on other components such as data-type sharing or inheritance; and
- component libraries that are simply lists of components are a special case of component libraries that incorporate component behaviour-specifications and / or metadata to aid in selection and adaptation.

Intractability results for these alternatives then follow from the observation that intractability results for a problem Π also hold for any problem Π' that has Π as a special case (suppose Π is intractable; if Π' is tractable by algorithm A, then A can be used to solve Π efficiently, which contradicts the intractability of Π — hence, Π' must also be intractable).

Our fp-tractability results are much more fragile, as innocuous changes to software requirements, software system, component, or component library structure may in fact violate assumptions critical to the operation of the algorithms underlying these results. Hence, they may only apply to certain more complex models, and this needs to be assessed on a case-by-case basis.

5.3 Discussion

We have found that designing and reconfiguring software systems by *de novo* design, component selection, and component selection with adaptation are all NP-hard (Result A). This implies that it is unlikely that deterministic polynomial-time methods exist for any of these problems. It also answers the question of which of *de novo* design, component selection, or component selection with adaptation is best for creating software systems – computationally speaking, all three methods are equally good (or bad) in general.

This NP-hardness has interesting additional implications. It is widely believed that $P = BPP$ [21, Section 5.2] where BPP is considered the most inclusive class of problems that can be efficiently solved using probabilistic methods (in particular, methods whose probability of correctness can be efficiently boosted to be arbitrarily close to probability one). Hence, our results also imply that unless $P = NP$, there are no probabilistic polynomial-time methods which correctly design or reconfigure software systems using the three approaches considered here with high probability for all inputs. Taken together, the above constitutes the first proof that *no* currently-used method (including search-based methods based on evolutionary algorithms (see [11, Section 5] and references) can guarantee both efficient and correct operation for all inputs for these problems.

As described in Section 4, efficient correctness-guaranteed methods may yet exist relative to plausible restrictions on the input and output. It seems reasonable to conjecture that some restrictions relative to the parameters listed in Table 1 should render these problems tractable. However, almost none of these restrictions or indeed many possible combinations of these restrictions can yield tractability, even when the parameters involved are restricted to very small constants (Results B–G). We do have some initial fp-tractability results (Results H and I). Taken together, our fp-results paint a disconcerting picture for problems SDes-Spec, SDes-CompA, SRec-Spec, and SRec-CompA, as they suggest that making both the software systems and the degree of allowable adaptation small under the approaches encoded in these problems does not yield tractability. However, the situation for problems SDes-Comp and SRec-Comp may not be so dire. To date, we have not been able to prove fp-intractability relative to a number of the individual parameters and parameter-combinations considered here for these problems. Hence, if fp-tractability holds in at least some of those cases, this would constitute proof that component selection without adaptation is tractable in a wider set of circumstances than either *de novo* design or component selection with adaptation.

Two valid objections to our fp-tractability results are that (1) the algorithms underlying these results are crude and rely on brute-force search and (2) the running times of these algorithms are (to be blunt, ludicrously) impractical. These objections are often true of the initial fp-algorithms derived relative to a parameter-set. However, once fp-tractability is proven, surprisingly effective fp-algorithms are often subsequently developed with greatly diminished non-polynomial terms and polynomial terms that are quadratic or even linear in the input size (see [4, 5] and references).

A final very important proviso is in order – namely, as illuminating as the results given here are in demonstrating basic forms of (in)tractability for software design and reconfiguration problems relative to the three approaches considered, these results do not necessarily imply that methods currently being applied to design or reconfigure software are impractical. Differing software and component models, the particular situations in which these currently-used methods are being applied, and accepted standards by which method practicality is assessed may render the results given here irrelevant. For example, current methods may already be implicitly exploiting restrictions on the input and output such that both efficient and correct operation (or operation that is correct with probability very close to one) are guaranteed. That being said, not knowing the precise conditions under which such practicality holds could have serious consequences, *e.g.*, drastically slowed software creation time and/or unreliable software operation, if these conditions are violated. These consequences would be particularly damaging in the case of runtime-based applications like ubiquitous and autonomic computing. Given that reliable software operation is very important and efficient software design and reconfiguration is at the very least desirable, the acquisition of such knowledge via a combination of rigorous empirical and theoretical analyses should be a priority. With respect to theoretical analyses, it is our hope that the techniques and results in this paper comprise a useful first step.

6. CONCLUSIONS

We have presented a formal characterization of the problems of software system design and reconfiguration by *de novo* design, component selection, and component selection with adaptation. Our complexity analyses reveal that, while these problems are computationally intractable in general, there are restrictions that render them tractable. Though the immediate practical utility of our tractability results is questionable given the basic models of software requirements, software systems and components relative to which they were derived, our intractability results give the first rigorous computational comparison between the three approaches to software design considered here. In particular, our results as a whole establish that all three approaches are equally computationally difficult in general but suggest that software creation by component selection may be tractable under more circumstances than the other two approaches.

There are several possible directions for future research. The first is to complete the analysis begun here to see if software creation by component selection really is fp-tractable for software systems that are small and/or created with small amounts of adaptation. The second is to extend this analysis to the more realistic models of software requirements, software systems, components, and component libraries sketched in Section 5.2. Last but certainly not least, derived tractability results should be implemented and tested in designing and reconfiguring actual software systems. Such testing will give invaluable guidance in phrasing future theoretical analyses along the lines sketched here; in turn, such analyses will hopefully help to guide future research in component-based software development.

7. ACKNOWLEDGMENTS

The authors extend most grateful thanks to Iris van Rooij and Maria Otworowska (Donders Institute for Brain, Cognition, and Behaviour, Radboud University Nijmegen) with whom they collaborated on the formalization of the adaptive toolbox theory of human decision-making that underlies the formalizations of component-based design and reconfiguration analyzed here. They would also like to thank the three reviewers for comments that helped greatly to improve the presentation of this paper. TW was supported by NSERC Discovery Grants RGPIN 228104-2010 and 228104-2015.

8. REFERENCES

[1] P. Allen and S. Frost. *Component-Based Development for Enterprise Systems: Applying the Select Perspective*. Cambridge University Press, 1997.

[2] R. G. Bartholet, D. C. Brogan, and P. F. Reynolds Jr. The computational complexity of component selection in simulation reuse. In *Proceedings of the 2005 Winter Simulation Conference*, pages 2472–2481, 2005.

[3] A. Brown. *Large-Scale Component-Based Software Development*. Prentice-Hall PTR, 2000.

[4] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015.

[5] R. Downey and M. Fellows. *Fundamentals of Parameterized Complexity*. Springer, Berlin, 2013.

[6] L. Fortnow. The Status of the P Versus NP Problem. *Communications of the ACM*, 52(9):78–86, 2009.

[7] W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE transactions on Software Engineering*, 21(7):529–536, 2005.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W.H. Freeman, 1979.

[9] G. Gigerenzer and P. Todd. Fast and frugal heuristics: The adaptive toolbox. In *Simple Heuristics that Make Us Smart*, pages 3–34. Oxford University Press, 1999.

[10] R. González and M. Torres. Critical issues in component-based development. In *Proceedings of The 3rd International Conference on Computing, Communications and Control Technologies*, 2005.

[11] M. Harman, S. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11, 2012.

[12] G. Heineman and B. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2000.

[13] H. Müller and N. Villegas. Runtime evolution of highly dynamic software. In *Evolving Software Systems*, pages 229–264. Springer, 2014.

[14] M. Nouri and J. Habibi. Approximating component selection with general costs. In *Advances in Computer Science and Engineering*, pages 61–68. Springer, 2008.

[15] M. Otworowska, M. Sweers, R. Wellner, M. Uhlmann, T. Wareham, and I. van Rooij. How did *Homo Heuristicus* become ecologically rational? In *Proceedings of the EuroAsianPacific Joint Conference on Cognitive Science*, pages 324–329. 2015.

[16] E. H. Page and J. M. Opper. Observations on the complexity of composable simulation. In *Proceedings of the 31st Winter Conference on Simulation*, volume 1, pages 553–560. ACM, 1999.

[17] M. Petty, E. Weisel, and R. Mielke. Computational complexity of selecting components for composition. In *Proceedings of the Fall 2003 Simulation Interoperability Workshop*, pages 14–19, 2003.

[18] U. Stege. The Impact of Parameterized Complexity to Interdisciplinary Problem Solving. In H. L. Bodlaender, R. Downey, F. V. Fomin, and D. Marx, editors, *The Multivariate Algorithmic Revolution and Beyond*, number 7370 in Lecture Notes in Computer Science, pages 56–68. Springer, Berlin, 2012.

[19] M. Sweers. *Adapting the Adaptive Toolbox: The Computational Cost of Building Rational Behaviour*. M.Sc. thesis, Radboud University Nijmegen, 2015.

[20] T. Wareham. *Systematic Parameterized Complexity Analysis in Computational Phonology*. Ph.D. thesis, University of Victoria, 1999.

[21] A. Wigderson. P, NP and mathematics — A computational complexity perspective. In *Proceedings of ICM 2006: Volume I*, pages 665–712, Zurich, 2007. EMS Publishing House.

APPENDIX

A. PROOFS OF SELECTED RESULTS

All of our intractability results will be derived using reductions from the following NP-hard problem:

DOMINATING SET [8, Problem GT2]
Input: An undirected graph $G = (V, E)$ and an integer k.
Question: Does G contain a dominating set of size $\leq k$, *i.e.*, is there a subset $V' \subseteq V$, $|V'| \geq k$, such that for all $v \in V'$, either $v \in V'$ or there is a $v' \in V'$ such that $(v, v') \in E$?

For each vertex $v \in V$, let the complete neighbourhood $N_C(v)$ of v be the set composed of v and the set of all vertices in G that are adjacent to v by a single edge, *i.e.*, $v \cup \{u \mid u \in V$ and (u, v) ∈ E$\}$. We assume below an arbitrary ordering on the vertices of V such that $V = \{v_1, v_2, \ldots, v_{|V|}\}$.

LEMMA 3. DOMINATING SET *polynomial-time many-one reduces to SDes-Spec such that in the constructed instance of SDes-Spec,* $|A| = 2$*,* $|sel| = 0$*, and* $|prc|$ *is a function of* k *in the given instance of* DOMINATING SET.

PROOF. Given an instance $\langle G = (V, E), k\rangle$ of DOMINATING SET, the constructed instance $\langle I, A, R, X = \langle |sel|, |prc|\rangle\rangle$ of SDes-Spec has $I = \{i_1, i_2, \ldots i_{|V|}\}$, $A = \{0, 1\}$, $|sel| = 0$, and $|prc| = k$. There are $|V| + 1$ situation-action pairs in R such that (1) for $r_i = (s_i, a_i)$, $1 \leq i \leq |V|$, $v(i_j) = True$ if $v_j \in N_C(v_i)$ and is $False$ otherwise and $a_i = 1$, and (2) for $r_{|V|+1} = (s_{|V|+1}, a_{|V|+1})$, all $v(i_j)$ are $False$ and $a_{|V|+1} = 0$. Note that the instance of SDes-Spec described above can be constructed in time polynomial in the size of the given instance of DOMINATING SET.

If there is a dominating set of size at most k in the given instance of DOMINATING SET, construct a software system consisting of a default `*`-condition selector and a single procedure in which the k `IF-THEN` statements have situation-variable conditions corresponding to the vertices in the dominating set (adding arbitrary other vertex situation-variables if the size of the dominating set is less than k) and action

1 and the final `ELSE` statement has action 0. Observe that this software system satisfies all situation-action pairs in R and has $|sel| = 0$ and $|prc| = k$.

Conversely, suppose that the constructed instance of SDes-Spec has a software system satisfying R with $|sel| = 0$ and $|prc| \leq k$. The selector in this system must be the default selector as it is the only selector with $|sel| = 0$. As for the single procedure attached to that selector, it has two possible structures:

1. *No negated situation-variable conditions*: As $r_{|V|+1}$ has no situation-variables set to $True$, it can only be processed correctly by the final `ELSE`-statement, which must thus execute action 0. In order to accept all other situation-pairs in R, the k conditions must then have situation-variables whose corresponding vertices form a dominating set in G of size at most k.
2. *Negated situation-variables are present*: Let c be the first negated situation-variable condition encountered moving down the code in the procedure, C be the set of unnegated situation-variable conditions encountered before c, and $R' \subseteq R$ be the set of situation-action pairs not recognized by the situation-variables in C. As $|prc| \leq k$, $|C| \leq k - 1$. Moreover, as situation-action pair $r_{|V|+1}$ has no situation-variable with value $True$ and hence cannot be recognized by an `IF-THEN` statement with an unnegated situation-variable condition, $r_{|V|+1} \in R'$.
   Let $R'' \subset R$ be such that $R' = R'' \cup \{r_{|V|+1}\}$. If R'' is empty, then the variables in C must have recognized all situation-action pairs in R except $r_{|V|+1}$, and hence the vertices corresponding to the situation-variables in C form a dominating set in G of size at most $k - 1$. If R'' is not empty, situation-variable c cannot have value $False$ for any of the situation-action pairs in R'' because having either 0 or 1 as the action executed by the associated `IF-THEN` statement would result in the procedure executing the wrong action for at least one situation-action pair in R' (if 0, at least one of the situation-action pairs in R''; if 1, $r_{|V|+1}$). However, this implies that all situation-action pairs in R'' have value $True$ for c, which in turn implies that the vertices corresponding to the situation-variables in $C \cup \{c\}$ form a dominating set in G of size at most k.

Hence, the existence of a satisfying software system for the constructed instance of SDes-Spec implies the existence of a dominating set of size at most k for the given instance of DOMINATING SET.

To complete the proof, note that in the constructed instance of SDes-Spec, $|A| = 2$, $|sel| = 1$, and $|prc| = k$. $\square$

LEMMA 4. DOMINATING SET *polynomial-time many-one reduces to SDes-Comp such that in the constructed instance of SDes-CompA, $|A| = 2$, $|L_{sel}| = 1$, and d is a function of k in the given instance of* DOMINATING SET.

PROOF. Given an instance $\langle G = (V, E), k\rangle$ of DOMINATING SET, the constructed instance $\langle I, A, R, L_{sel}, L_{prc}, d\rangle$ of SDes-Comp has $I = \{i_1, i_2, \ldots i_{2|V|}\}$ and $A = \{0, 1\}$. There are $|V| + 1$ situation-action pairs in R such that (1) for $r_i = (s_i, a_i)$, $1 \leq i \leq |V|$, $v(i_i) = True$, $v(i_{|V|+j}) = True$ if $v_j \in N_C(v_i)$, all other $v(i_j)$ are $False$, and $a_i = 1$, and (2) for $r_{|V|+1} = (s_{|V|+1}, a_{|V|+1})$, all $v(i_j) = False$ and $a_{|V|+1} = 0$. There is a single selector in L_{sel} whose $|V| - 1$ `IF-THEN` statement conditions are the first $|V| - 1$ situation-variables in I. There are $|V|$ procedures in L_{prc} such that procedure i, $1 \leq i \leq |V|$, has an `IF-THEN` statement for each of the vertices in $N_C(v_i)$ whose condition is the situation-variable associated with that vertex and which executes action 1 and an `ELSE` statement that executes action 0. Finally, let $d = k+1$. Note that the instance of SDes-Comp described above can be constructed in time polynomial in the size of the given instance of DOMINATING SET.

If there is a dominating set of size at most k in the given instance of DOMINATING SET, construct a software system consisting of the single selector in L_{sel} and the at most k procedures from L_{prc} corresponding to the vertices in the dominating set. For each of the $|V| - 1$ `IF-THEN` statements in the selector, call the procedure encoding the neighbourhood of the vertex in the dominating set which dominates the vertex corresponding to the situation-variable in that statement's condition. For the `ELSE` statement in the selector, execute the procedure encoding the neighbourhood of the vertex in the dominating set that dominates vertex $v_{|V|}$. Observe that this software system satisfies all situation-action pairs in R and has $d = k + 1$.

Conversely, suppose that the constructed instance SDes-Comp has a software system constructed from at most $d = k + 1$ distinct components from L_{sel} and L_{prc} that satisfies all of the situation-action pairs in R. As the selector is constructed such that each of the first $|V|$ situation-action pairs in R has its own associated procedure and each procedure accepts the vertices in a specific vertex-neighbourhood in G, in order for these situation-action pairs to be satisfied, the distinct procedures in the software system must correspond to a dominating set for G (note that both $r_{|V|}$ and $r_{|V|+1}$ are satisfied by the procedure associated with the `ELSE` statement in the selector). As $d = k + 1$ and the system had to incorporate a selector from L_{sel}, this means that there are at most k such procedures and hence a dominating set of size at most k in the given instance of DOMINATING SET.

To complete the proof, note that in the constructed instance of SDes-Comp, $|A| = 2$, $|L_{sel}| = 1$, and $d = k + 1$. $\square$

Result A: *SDes-Spec and SDes-Comp are NP-hard.*

PROOF. Follows from the NP-hardness of DOMINATING SET and the reductions in Lemmas 3 and 4, respectively. $\square$

Result B: *SDes-Spec is $W[2]$-hard for $\{|A|, |sel|, |prc|\}$ when $|A| = 2$ and $|sel| = 0$.*

PROOF. Follows from the $W[2]$-hardness of $\{k\}$-DOMINATING SET [5] and the reductions from DOMINATING SET to SDes-Spec given in Lemma 3. $\square$