CS-TR-4997

Specifying and Verifying the Correctness of Dynamic Software Updates[⋆]

Christopher M. Hayden[1], Stephen Magill[1], Michael Hicks[1],
Nate Foster[2], and Jeffrey S. Foster[1]

[1] Computer Science Department, University of Maryland, College Park
{hayden,smagill,mwh,jfoster}@cs.umd.edu
[2] Computer Science Department, Cornell University
jnfoster@cs.cornell.edu

Abstract. Dynamic software updating (DSU) systems allow running programs to be patched on-the-fly to add features or fix bugs. While dynamic updates can be tricky to write, techniques for establishing their correctness have received little attention. In this paper, we present the first methodology for automatically verifying the correctness of dynamic updates. Programmers express the desired properties of an updated execution using *client-oriented specifications* (CO-specs), which can describe a wide range of client-visible behaviors. We verify CO-specs automatically by using off-the-shelf tools to analyze a *merged* program, which is a combination of the old and new versions of a program. We formalize the merging transformation and prove it correct. We have implemented a program merger for C, and applied it to updates for the Redis key-value store and several synthetic programs. Using Thor, a verification tool, we could verify many of the synthetic programs; using Otter, a symbolic executor, we could analyze every program, often in less than a minute. Both tools were able to detect faulty patches and incurred only a factor-of-four slowdown, on average, compared to single version programs.

1 Introduction

Dynamic software updating (DSU) systems allow programs to be patched on-the-fly, to add features or fix bugs without incurring downtime. DSU systems were originally developed for a few limited domains such as telecommunications networks, financial transaction processors, and the like, but are now becoming pervasive. Ksplice, recently acquired by Oracle, supports applying Linux kernel security patches dynamically [15]. The Erlang language, which provides built-in support for dynamic updates, is gaining in popularity for building server programs [2]. Many web applications employ DSU techniques to provide 24/7

[⋆] University of Maryland, Department of Computer Science Technical Report CS-TR-4997. This paper is an extended version of the paper of the same title appearing in *Verified Software: Theories, Tools and Experiments (VSTTE) 2012.* This version includes additional discussion and proofs in the appendices.

service to a global audience—for these systems, there is no single time of day when taking down the service to perform upgrades is acceptable.

Given the increasing need for DSU, a natural question is: How can developers ensure a dynamically updated program will behave correctly? Today, developers need to reason manually about all the pieces of an updating program: the old program version, the new program version, and code that transforms the state of the (old) running version into the form expected by the new version. Moreover, they need to repeat this reasoning process for each allowable "update point" during execution. In our experience this is a tricky proposition in which it is all too easy to make mistakes. Despite such difficulties, most DSU systems do not address the issue of correctness, or they focus exclusively on generic safety properties, such as type safety, that rule out obviously wrong behavior [7, 22–24] but are insufficient for establishing correctness [11].

This paper presents a methodology for verifying the correctness of dynamic updates. Rather than propose a new verification algorithm that accounts for the semantics of updating, we develop a novel program transformation that produces a program suitable for verification with off-the-shelf tools. Our transformation *merges* an old program and an update into a program that simulates running the program and applying the update at any allowable point. We formalize our transformation and prove that it is correct (Section 3).

We are particularly interested in using our transformation to prove execution properties from clients' points of view, to show that a dynamic update does not disrupt active sessions. For example, suppose we wish to update a key-value store such as Redis [20] so that it uses a different internal data structure. To verify this update's transformation code, we could prove that values inserted into the store by the client are still present after it is dynamically updated. We call such specifications *client-oriented specifications* (or *CO-specs* for short).

We have identified three categories of CO-specs that capture most properties of interest: *backward-compatible* CO-specs describe properties that are identical in the old and new versions; *post-update* CO-specs describe properties that hold after new features are added or bugs are fixed by an update; and *conformable* CO-specs describe properties that are identical in the old and new versions, modulo uniform changes to the external interface. CO-specs in these categories can often be mechanically constructed from CO-specs written for either the old or new program alone. Thus, if a programmer is inclined to verify each program version using CO-specs, there is little additional work to verify a dynamic update between the two. Nevertheless, some interesting and subtle properties lie outside these categories, so our framework also allows arbitrary properties to be expressed (Section 2).

We have implemented our merging transformation for C programs and used it in combination with two existing tools to verify properties of several dynamic updates (Section 4). We chose the symbolic executor Otter [21] and the verification tool Thor [16] as they represent two ends of the design space: symbolic execution is easy to use and scales reasonably well but is incomplete, while verification scales less well but provides greater assurance. We wrote two synthetic benchmarks, a key-value store and a multiset implementation, and designed dynamic patches for them based on realistic changes (e.g., one change was inspired by an update to the storage server Cassandra [5]). We also wrote dynamic patches for six releases of Redis [20], a popular, open-source key-value store. We used the Redis code as is, and wrote the state transformation code ourselves.

We checked all the benchmark programs with Otter and verified several properties of the synthetic updates using Thor. Both tools successfully uncovered bugs that were intentionally and unintentionally introduced in the state transformation code. The running time for verification of merged programs was roughly four times slower than single-version checking. This slowdown was due to the additional branching introduced by update points and the need to analyze the state transformer code. As tools become faster and more effective, our approach will scale with them. In summary, this paper makes three main contributions:

- It presents the first automated technique for verifying the behavioral correctness of dynamic updates.
- It proposes *client-oriented specifications* as a means to specify general update correctness properties.
- It shows the effectiveness of merging-based verification on practical examples, including Redis [20], a widely deployed server program.

2 Defining dynamic software update correctness

Before we can set out verifying DSU correctness, we have to decide what correctness is. In this section, we first review previously proposed notions of correctness and argue why they are insufficient for our purposes. Then we propose *client-oriented specifications* (CO-specs) as a means of specifying correctness properties, and argue that this notion overcomes limitations of prior notions. We also describe a simple refactoring that allows CO-specs to be used to verify client-server programs that communicate over a network.

2.1 Prior work on update correctness

Kramer and Magee [14] proposed that updates are correct if they are *observationally equivalent*—i.e., if the updated program preserves all observable behaviors of the old program. Bloom and Day [3] observed that, while intuitive, this is too restrictive: an update may fix bugs or add new features.

To address the limitations of strict observational equivalence, Gupta et al. [9] proposed *reachability.* This condition classifies an update as correct if, after the update is applied, the program eventually *reaches* some state of the new program. Reachability thus admits bugfixes, where the new state consists of the corrected code and data, as well as feature additions, where the new state is the old data plus the new code and any new data. Unfortunately, reachability is both too permissive and too restrictive, as shown by the following example. Version 1.1.2 of the `vsftpd` FTP server introduced a feature that limits the number of connections from a single host. If we update a running vsftpd server, we would expect it to preserve any active connections. But doing so violates reachability. If the number of connections from a particular host exceeds the limit and these connections remain open indefinitely, the server will never enter a reachable state of the new program. On the other hand, reachability would allow an update that terminates all existing connections. This is almost certainly not what we want—if we were willing to drop existing connections we could just restart the server!

We believe that the flaw in all of these approaches is that they attempt to define correctness in a completely general way. We think it makes more sense for programmers to specify the behavior they expect as a collection of properties. Some properties will apply to multiple versions of the program while other properties will change as the program evolves. Because the goal of a dynamic update is to preserve active processing and state, the properties should express the expected continuity that a dynamic update is meant to provide to active clients. We therefore introduce *client-oriented specifications* (CO-specs) to specify update properties that satisfy these requirements.

2.2 Client-oriented specifications

We can think of a CO-spec as a kind of client program that opens connections, sends messages, and asserts that the output received is correct. CO-specs resemble tests, but certain elements of the test code are left abstract for generality (cf. Figure 1). For example, consider again reasoning about updates to a key-value store such as Redis. A CO-spec might model a client that inserts a key-value pair into the store and then looks up the key, checking that it maps to the correct value (even if a dynamic update has occurred in the meantime). We can make such a CO-spec general by leaving certain elements like the particular keys or values used unconstrained. Similarly, we can allow arbitrary actions to be interleaved between the insert and lookup. Such specifications capture essentially arbitrary client interactions with the server.

Our goal is to use our program transformation, defined in Section 3, to produce a *merged* program that we can verify using off-the-shelf tools. But existing tools only verify single programs in isolation, so we cannot literally write CO-specs as client programs that communicate with a server being updated. To verify a CO-spec in a real client-server program we replace the server's main function the CO-spec and call the relevant server functions directly. In doing so, we are checking the server's core functionality, but not its main loop or any networking code. For example, suppose our key-value store implements functions get and set to read and write mappings from the store, and the server's main loop would normally dispatch to these functions. CO-specs would call the functions directly as shown in Figure 1. Here, ? denotes a non-deterministically chosen (integer) value, and assume and assert have their standard semantics. If updates are permitted while executing either get or set, verifying Figure 1(b) will establish that the assertions at the end of the specification hold no matter when the update takes place.

```
int get(int k, int *v);
void set(int k, int v);

void arbitrary (int k1) {
 int k2 = ?, v = ?;
 if (k1 == k2 || ?)
   get(k2,&v);
 else set (k2,v);
}
```

(a) interface, helper

```
void back_compat_spec() {
  int k = ?, v_in = ?;
  int v_out, found;
  set(k, v_in );
  while(?) arbitrary (k);
  found = get(k,&v_out);
  assert (found &&
          v_out == v_in);
}
```

(b) backward-compat. spec

```
void post_update_spec() {
  int k = ?;
  int v_out, found;
  while(?) arbitrary (?);
  assume(is_updated);
  delete(k);
  found = get(k,&v_out);
  assert (!found);
}
```

(c) post-update spec

Fig. 1. Sample C specifications for key-value store.

In our experience writing CO-specs for updates, we have found that they often fall into one of the following categories:

- *Backward-compatible CO-specs* describe behaviors that are unaffected by an update. For the data structure-changing update to Redis mentioned earlier, the CO-spec in Figure 1(b) would check that existing mappings are preserved.
- *Post-update CO-specs* describe behavior specific to the new program version. For example, suppose we added a delete feature to the key-value store. Then the CO-spec in Figure 1(c) verifies that, after the update, the feature is working properly. The CO-spec employs the flag is_updated, which is true after an update has taken place, to ensure that we are testing the new or changed functionality after the update. We discuss the semantics of this flag in Section 3.
- *Conformable CO-specs* describe updates that change interfaces, but preserve core functionality. For example, suppose we added namespaces to our key-value store, so that get and set take an additional namespace argument. The state transformation code would map existing entries to a default namespace. A conformable CO-spec could check that mappings inserted prior to the update are present in the default namespace afterward; in essence, the CO-spec would associate old-version calls with new-version calls at the default namespace. (Further details are given in Appendix A.3.)

These categories encompass prior notions of correctness. Backward compatible specifications capture the spirit of Kramer and Magee's condition, but apply to individual, not all, behaviors. The combination of backward-compatible and post-update specifications capture Bloom and Day's notions of "future-only implementations" and "invisible extensions"—parts of a program whose semantics change but not in a way that affects existing clients [3]. The combination of backward-compatible and conformable specifications match ideas proposed by Ajmani et al. [1], who studied dynamic updates for distributed systems and proposed mechanisms to maintain continuity for clients of a particular version.

CO-specs can also be used to express the constraints intended by Gupta's *reachability* while side-stepping the problem that reachability can leave behavior under-constrained. For example, for the vsftpd update mentioned above, the programmer can directly write a CO-spec that expresses what should happen to existing client connections, e.g., whether all, some, or none should be preserved. This does not fall into one of the categories above, demonstrating the utility of a full specification language over "one size fits all" notions of update correctness.

Another feature of CO-specs in these categories is that they can be mechanically constructed from CO-specs that are written for a single version. Thus, if a programmer was inclined to verify the correctness of each version of his program using CO-specs, the additional work to verify a dynamic update is not much greater. For details, see Appendix A.

3 Verification via program merging

We verify CO-specs by *merging* an existing program version with its update, so that the semantics of the merged program is equivalent to the updating program. This section formalizes a semantics for dynamic updates to single-threaded programs, then defines the merging transformation and proves it correct with respect to the semantics. Many server programs for which dynamic updating is useful are single-threaded [12, 18, 11]. However, an important next step for this work would be to adapt it to support updates to multi-threaded (and distributed) programs.

3.1 Syntax

The top of Figure 2 defines the syntax of a simple programming language supporting dynamic updates. It is based on the Proteus dynamic update calculus [22], and closely models the semantics of common DSU systems, including Ginseng [18] (which is the foundation of our implementation), Ksplice [15], Jvolve [23], K42 [13], DLpop [12], Dynamic ML [24] and Bracha's DSU system [4].

A *program* p is a mapping from function names g to functions $\lambda x.e$. A function body e is defined by a mostly standard core language with a few extensions for updating. Our language contains a construct update, which indicates a position where a dynamic update may take effect. To support writing specifications, the language includes an expression ?, which represents a random integer, and expressions assume v, assert v, and running p, all of whose semantics are discussed below. Expressions are in administrative normal (A-normal) form [8] to keep the semantics simple—e.g., instead of e_1+e_2, we write $\text{let } x = e_1 \text{ in let } y = e_2 \text{ in } x+y$. We write $e_1; e_2$ as shorthand for $\text{let } x = e_1 \text{ in } e_2$, where x is fresh for e_2.

3.2 Semantics

The semantics, given in the latter half of Figure 2, is written as a series of small-step rewriting rules between *configurations* of the form $\langle p; \sigma; e\rangle$, which contain the program p, its current heap σ, and the current expression e being evaluated. A heap is a partial function from locations l to values v, and a location l is either

$$
\begin{array}{lll}
\textit{Prog.} & p & ::= p, (g, \lambda x.e) \mid \cdot \\
\textit{Exprs.} & e & ::= v \mid v_1\ \mathit{op}\ v_2 \mid v_1(v_2) \mid\ ?\ \mid\ !v \mid \mathsf{ref}\ v \mid \\
 & & \quad v_1 := v_2 \mid \mathsf{if}\ v\ e_1\ e_2 \mid \mathsf{update} \mid \\
 & & \quad \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{assume}\ v \mid \\
 & & \quad \mathsf{while}\ e_1\ \mathsf{do}\ e_2 \mid \mathsf{assert}\ v \mid \\
 & & \quad \mathsf{running}\ p \mid \mathsf{error} \\
\textit{Values} & v & ::= x \mid l \mid i \mid (v_1, v_2) \mid () \\
\textit{Locs.} & l & ::= a \mid g
\end{array}
\qquad
\begin{array}{ll}
\textit{Variables} & x, y, z \\
\textit{Globals} & f, g \\
\textit{Operators} & \mathit{op} \\
\textit{Integers} & i, j \\
\textit{Addresses} & a \\
\textit{Heaps} & \sigma \in \textit{Locs} \rightharpoonup \textit{Values} \\
\textit{Patch} & \pi ::= (p, e) \\
\textit{Labels} & \nu ::= \pi \mid \epsilon
\end{array}
$$

$$
\begin{array}{lll}
\langle p; \sigma; v_1\ \mathit{op}\ v_2 \rangle & \rightsquigarrow \langle p; \sigma; v' \rangle & v' = [\![\mathit{op}]\!](v_1, v_2) \\
\langle p; \sigma; \mathsf{ref}\ v \rangle & \rightsquigarrow \langle p; \sigma[a \mapsto v]; a \rangle & a \notin \mathit{dom}(\sigma) \\
\langle p; \sigma; !l \rangle & \rightsquigarrow \langle p; \sigma; v \rangle & \sigma(l) = v \text{ and } l \notin \mathit{dom}(p) \\
\langle p; \sigma; a := v \rangle & \rightsquigarrow \langle p; \sigma[a \mapsto v]; v \rangle & a \in \mathit{dom}(\sigma) \\
\langle p; \sigma; g := v \rangle & \rightsquigarrow \langle p; \sigma[g \mapsto v]; v \rangle & g \notin \mathit{dom}(p) \\
\langle p; \sigma; ? \rangle & \rightsquigarrow \langle p; \sigma; i \rangle & \text{for some } i \\
\langle p; \sigma; \mathsf{let}\ x = v\ \mathsf{in}\ e \rangle & \rightsquigarrow \langle p; \sigma; e[v/x] \rangle & \\
\langle p; \sigma; f(v) \rangle & \rightsquigarrow \langle p; \sigma; e[v/x] \rangle & p(f) = \lambda x.e \\
\langle p; \sigma; \mathsf{if}\ 0\ e_1\ e_2 \rangle & \rightsquigarrow \langle p; \sigma; e_2 \rangle & \\
\langle p; \sigma; \mathsf{if}\ v\ e_1\ e_2 \rangle & \rightsquigarrow \langle p; \sigma; e_1 \rangle & v \neq 0 \\
\langle p; \sigma; \mathsf{while}\ e_1\ \mathsf{do}\ e_2 \rangle & \rightsquigarrow \langle p; \sigma; \mathsf{let}\ x = e_1\ \mathsf{in} & x \notin \mathit{fv}(e_1, e_2) \\
 & \qquad \mathsf{if}\ x\ (e_2; \mathsf{while}\ e_1\ \mathsf{do}\ e_2)\ 0 \rangle & \\
\langle p; \sigma; \mathsf{update} \rangle & \rightsquigarrow \langle p; \sigma; 0 \rangle & \\
\langle p; \sigma; \mathsf{update} \rangle & \overset{\pi}{\rightsquigarrow} \langle p_\pi; \sigma; (e_\pi; 1) \rangle & \pi = (p_\pi, e_\pi) \\
\langle p; \sigma; \mathsf{running}\ p \rangle & \rightsquigarrow \langle p; \sigma; 1 \rangle & \\
\langle p; \sigma; \mathsf{running}\ p' \rangle & \rightsquigarrow \langle p; \sigma; 0 \rangle & p' \neq p \\
\langle p; \sigma; \mathsf{assume}\ v \rangle & \rightsquigarrow \langle p; \sigma; v \rangle & v \neq 0 \\
\langle p; \sigma; \mathsf{assert}\ v \rangle & \rightsquigarrow \langle p; \sigma; v \rangle & v \neq 0 \\
\langle p; \sigma; \mathsf{assert}\ 0 \rangle & \rightsquigarrow \langle p; \sigma; \mathsf{error} \rangle & \\
\langle p; \sigma; \mathsf{let}\ x = \mathsf{error}\ \mathsf{in}\ e \rangle & \rightsquigarrow \langle p; \sigma; \mathsf{error} \rangle &
\end{array}
$$

$$
\frac{\langle p; \sigma; e_1 \rangle \overset{\nu}{\rightsquigarrow} \langle p'; \sigma'; e_1' \rangle}{\langle p; \sigma; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rangle \overset{\nu}{\rightsquigarrow} \langle p'; \sigma'; \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2 \rangle}
$$

Fig. 2. Syntax and semantics.

a (dynamically allocated) address a or a (static) global name g. Note that while the language does not include closures, global names g are values, and so the language does support C-style function pointers.[1]

Most of the operational semantics rules are straightforward. We write $e[x/v]$ for the capture-avoiding substitution of x with v in e. We assume that the semantics of primitive operations op is defined by some mathematical function $[\![op]\!]$; e.g., $[\![+]\!]$ is the integer addition function. Loops are rewritten to condition-

[1] Variables names x are values so that we can use a simple grammar to enforce A-normal form. The downside is that syntactically well-formed programs could pass around unbound variables and store them in the heap. The ability to express such programs is immaterial to our modeling of DSU, and could be easily ruled out with a simple static type system.

als, where in both cases a non-zero guard is treated as true and zero is treated as false. Addresses a for dynamically allocated memory must be allocated prior to assigning to them, whereas a global variable g is created when it is first assigned to. This semantics allows state transformation functions, described below, to define new global variables that are accessible to an updated program.

The update command identifies a position in the program at which a dynamic update may take place. Semantically, update non-deterministically transitions either to 0, indicating that an update did not occur, or to 1 (eventually), indicating that a dynamic update was available and was applied.[2] In the case where an update occurs, the transition arrow is labeled with the patch π; all other (unadorned) transitions implicitly have label ϵ. A patch π is a pair (p_π, e_π) consisting of the new program code (including unmodified functions) p_π and an expression e_π that transforms the current heap as necessary, e.g., to update an existing data structure or add a new one for compatibility with the new program p_π. In practice, e_π will be a call to a function defined in p_π. The transformer expression e_π is placed in redex position and is evaluated immediately; to avoid capture, non-global variables may not appear free in e_π. Notice that an update that changes function f has no effect on running instances of f since evaluation of their code began prior to the update taking place.

The placement of the update command has a strong influence on the semantics of updates. Placing update pervasively throughout the code essentially models asynchronous updates. Or, as prior work recommends [14, 1, 18, 11], we could insert update selectively, e.g., at the end of each request-handling function or within the request-handling loop, to make an update easier to reason about

The constructs $\mathsf{running}\ p$, $\mathsf{assume}\ v$, and $\mathsf{assert}\ v$ allow us to write specifications. The expression $\mathsf{running}\ p$ returns 1 if p is the program currently running and 0 otherwise; i.e., we encode a program version as the program text itself. (In Figure 1(c) the expression $\mathsf{is\_updated}$ is equivalent to $\mathsf{running}\ p$ where p is the new program version.) The expression $\mathsf{assert}\ v$ returns v if it is non-zero, and error otherwise, which by the rule for let propagates to the top level. Finally, the expression $\mathsf{assume}\ v$ returns v if v is non-zero, and otherwise is stuck.

3.3 Program merging transformation

We now present our program merging transformation, which takes an old program configuration $\langle p, \sigma, e\rangle$ and a patch π and yields a single *merged program* configuration, written $\langle p, \sigma, e\rangle \rhd \pi$. We present the transformation formally and then prove that the merged program is equivalent to the original program with the patch applied dynamically. While we focus on merging a program with a single update, the merging strategy can be readily generalized to multiple updates (we sketch the generalization in Appendix B).

The definition of $\langle p, \sigma, e\rangle \rhd \pi$ is given in Figure 3(e). It makes use of functions $[\![\cdot]\!]^{\cdot}$ and $\{\!|\cdot|\!\}^{\cdot}$, defined in Figure 3(a)–(d). We present the interesting cases; the

[2] In practice, update would be implemented by having the run-time system check for an update and apply it if one is available [12].

$$
\begin{aligned}
&\llbracket p', (g, \lambda y.e) \rrbracket^{p,\pi} \triangleq \\
&\qquad \llbracket p' \rrbracket^{p,\pi},\ (g, \lambda y.\llbracket e \rrbracket^{p,\pi}), \\
&\qquad (g_{ptr}, \lambda y.\mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ g'(y)\ g(y)) \\
&\llbracket \cdot \rrbracket^{p,\pi} \triangleq (\cdot, (\mathit{isupd}, \lambda y.\mathsf{let}\ z = !\mathit{uflag}\ \mathsf{in}\ z > 0))
\end{aligned}
$$

(a) Old version programs

$$
\begin{aligned}
&\{\!| p', (g, \lambda y.e) |\!\}^{p} \triangleq \\
&\qquad \{\!| p' |\!\}^{p}, (g', \lambda y.\{\!| e |\!\}^{p}) \\
&\{\!| \cdot |\!\}^{p} \triangleq \cdot
\end{aligned}
$$

(b) New version programs

$$
\begin{aligned}
&\llbracket g \rrbracket^{p,\pi} \triangleq \\
&\qquad \begin{cases} g_{ptr} & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases} \\
&\llbracket \mathsf{running}\ p'' \rrbracket^{p,(p_\pi,e_\pi)} \triangleq \\
&\qquad \begin{cases} \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ z = 0 & \text{if } p = p'' \\ \mathit{isupd}() & \text{if } p_\pi = p'' \\ 0 & \text{otherwise} \end{cases} \\
&\llbracket \mathsf{update} \rrbracket^{p,(p_\pi,e_\pi)} \triangleq \\
&\qquad \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} \\
&\qquad \mathsf{if}\ z\ 0\ (\mathit{uflag} := ?; \\
&\qquad\qquad \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!| e |\!\}^{p_\pi}; 1)\ 0)
\end{aligned}
$$

(c) Old version expressions

$$
\begin{aligned}
&\{\!| g |\!\}^{p} \triangleq \\
&\qquad \begin{cases} g' & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases} \\
&\{\!| \mathsf{running}\ p'' |\!\}^{p} \triangleq \\
&\qquad \begin{cases} 1 & \text{if } p = p'' \\ 0 & \text{otherwise} \end{cases} \\
&\{\!| \mathsf{update} |\!\}^{p} \triangleq 0
\end{aligned}
$$

(d) New version expressions

$$
\begin{aligned}
&\langle p; \sigma; e \rangle\ \rhd\ \pi \triangleq \langle \overline{p}, \overline{\sigma}[\mathit{uflag} \mapsto i], \overline{e} \rangle \\
&\qquad \text{where } (p_\pi, e_\pi) = \pi \qquad \overline{p} = \{\!| p_\pi |\!\}^{p_\pi}, \llbracket p \rrbracket^{p,\pi} \qquad \overline{e} = \llbracket e \rrbracket^{p,\pi} \\
&\qquad\qquad i \leq 0 \qquad \overline{\sigma} = \{ l \mapsto \llbracket v \rrbracket^{p,\pi} \mid \sigma(l) = v \}
\end{aligned}
$$

(e) Merging a configuration and a patch

Fig. 3. Merging transformation (partial).

remaining cases are translated structurally in the natural way. For simplicity, the transformation assumes the updated program p_π does not delete any functions in p. Deletion of function f can be modeled by a new version of f with the same signature as the original and the body $\mathsf{assert(0)}$.

The merging transformation renames each new-version function from g to g', and changes all new-version code to call g' instead of g (the first rewrite rules in Figure 3(b) and (d), respectively). For each old-version function g, it generates a new function g_{ptr} whose body conditionally calls the old or new version of g, depending on whether an update has occurred (Figure 3(a)). The transformation introduces a global variable *uflag* (Figure 3(e)) and a function *isupd* to keep track of whether the update has taken place (bottom of Figure 3(a)). All calls to g in the old version are rewritten to call g_{ptr} instead (top of Figure 3(c)).

The transformation rewrites occurrences of update in old-version code into expressions that check whether *uflag* is positive (bottom of Figure 3(c)). If it is, then the update has already taken place, so there is nothing to do. Otherwise, the transformation sets *uflag* to ?, which simulates a non-deterministic choice of whether to apply the update. If *uflag* now has a positive value, the update path was chosen, so the transformation executes the developer-provided state transformation e, which must also be transformed according to $\{\!|\cdot|\!\}^{\cdot}$ to properly reference functions in the new program. While this transformation results in multiple occurrences of the expression e, in practice e is a call to a state transformation function defined in the new version and so does not significantly increase code size.

Version tests $\mathsf{running}\ p$ are translated into calls to *isupd* in the old version, and to appropriate constants in the new code (since we know the update has occurred if new code is running).

3.4 Equivalence

We can now prove that an update to an old-program configuration is correct if and only if the result of merging that configuration and the update is correct. This result lets us use stock verification tools to check properties of dynamic updates using the merged program, which simulates updating, instead of having to develop new tools or extend existing ones.

We say that a program and a sequence of updates are *correct* if evaluation never reaches error (i.e., if there are no assertion failures). More formally:

Definition 1 (Correctness) *A configuration* $\langle p; \sigma; e\rangle$ *and an update* π *are* correct*, written* $\models \langle p; \sigma; e\rangle, \pi$*, if and only if for all* p', σ', e' *it is the case that* $\langle p; \sigma; e\rangle \overset{\pi}{\rightsquigarrow}{}^{*} \langle p'; \sigma'; e'\rangle$ *implies* e' *is not* error*.*

The expression e at startup could be a call to an entry-point function (i.e., main). A correct program need not apply π, though no other update may occur. When no update is permitted we write $\models \langle p; \sigma; e\rangle$.

Theorem 1 (Equivalence) *For all* p, σ, e, π *such that* $dom(p_\pi) \supseteq dom(p)$ *we have that* $\models \langle p; \sigma; e\rangle, \pi$ if and only if $\models (\langle p, \sigma, e\rangle \rhd \pi)$*.*

The proof is by bisimulation and is given in Appendix C along with proof sketches of key supporting lemmas.

Observe that type errors result in stuck programs, e.g., !1 does not reduce, while the above theorem speaks only about reductions to error. We have chosen not to consider type safety in the formal system to keep things simple; adding types, we could appeal to standard techniques [22–24, 7]. Our implementation catches type errors that could arise due to a dynamic update by transforming them into assertion violations. In particular, we rename functions and global variables whose type has changed prior to merging, essentially modeling the change as a deletion of one variable and the addition of another. Deleted functions are modeled as mentioned above, and deleted global variables are essentially assigned the error expression. Thus, any old code that accesses a stale definition post-update (including one with a changed type) fails with an assertion violation.

4 Experiments

To evaluate our approach, we have implemented the merging transformation for C programs, with the additional work to handle C being largely routine. We merged several programs and dynamic updates and then checked the merged programs against a range of CO-specs. We analyzed the merged programs using two different tools: the symbolic executor Otter, developed by Ma et al. [21], and the verification tool Thor, developed by Magill et al. [17]. The tools represent a tradeoff: Otter is easier to use and more scalable but provides incomplete assurance, while Thor can guarantee correctness but is less scalable and requires more manual effort. Overall, both tools proved useful. Otter successfully checked all the COs-specs we tried, generally in less than one minute. Thor was able to fully verify several updates, though running times were longer. Both tools found bugs in updates, including mistakes we introduced inadvertently. On average, verification of merged code took four times longer than verification of a single version. Since our approach is independent of the verification tool used, its performance and effectiveness will improve as advances are made in verification technology.

4.1 Programs

We ran Otter and Thor on updates to three target programs. The first two are small, synthetic examples: a multiset server, which maintains a multiset of integer values, and a key-value store. For each program, we also developed a number of updates inspired by common program changes such as memory and performance optimizations and semantic changes observed in real-world systems such as Cassandra [5]. The third program we considered is Redis [20], a widely used open-source key-value server. At roughly 12k lines of C code, Redis is significantly larger that our synthetic examples, and is currently not tractable for Thor. We developed six dynamic patches for Redis that update between each pair of consecutive versions from 1.3.6 through 1.3.12, and we also wrote a set of CO-specs that describe basic correctness properties of the updates.

As we mention in Section 2, we join each CO-spec with the server code and have the main function invoke the CO-spec after it initializes server data structures. The new-version source code includes the state transformation code, which is identified by a distinguished function name recognized by the merger.

Synthetic Examples. Figure 4 lists the synthetic benchmarks we constructed for our multiset and key-value store programs. Each grouping of rows shows a dynamic update and a list of CO-specs we wrote for that update. The multiset program has routines to add and delete elements and to test membership. The updates both change to a set semantics, where duplicate elements are disallowed. The first (correct) state transformer removes all duplicates from a linked list that maintains the current multiset. The second update has a broken state transformer that fails to remove duplicates.

The key-value store program also implements its store with a linked list. The updates are inspired by code changes we have seen in practice and include a bug

Program – *change*	**Thor time (s)**			**Otter time (s)**		
CO-specs	**old**	**new**	**mrg**	**old**	**new**	**mrg**
Multiset – *disallow duplicates* (correct)						
mem-mem^b	90.11	121.27	1003.22	6.29	9.72	49.37
add-mem^b	64.17	89.71	537.01	3.26	10.48	50.84
add-add-del-set^g			–			4.04
Multiset – *disallow duplicates* (broken)						
mem-mem^b	25.33	57.78	133.68	6.28	9.77	42.5
add-mem^b	15.68	33.50	80.07	3.25	9.94	33.53
add-add-del-set-fails^g			122.71			5.49
Key-value store – *bug fix*						
put-get^b	27.01	26.13	41.62	3.28	2.54	18.42
new-def-shadows^g			–			4.19
new-def-shadows-bc-fails^b	38.97	41.52	117.56	3.88	2.06	19.03
Key-value store – *added namespaces*						
new-def-shadows-post^p		–	–		1.02	2.99
put-get^p		–	–		18.32	228.69
new-def-shadows-conf^c	–	–	–	1.19	1.93	7.53
put-get-conf^c	–	–	–	4.23	7.09	61.41
Key-value store – *optimization* (broken)						
put-get-back^b	42.133	–	–	2.08	11.01	56.44
new-def-shadows-back^b	15.344	–	–	2.14	11.33	56.03
Key-value store – *optimization* (correct)						
put-get-back^b	41.87	–	–	2.07	10.87	69.31
new-def-shadows-back^b	15.72	–	–	2.14	10.96	68.95

b – backward compatible p – post update c – conformable g – general
A dash indicates that the example could not be verified.

Fig. 4. Synthetic examples.

fix (bindings could not be overwritten), a feature addition (adding namespaces), and an optimization (removing overwritten bindings), where for this last update the state transformer was broken at first.

The properties span all the categories of CO-specs that we outlined in Section 2. Backward compatible specs, such as add-mem, check core functionality that does not change between versions (add actually adds elements, delete removes elements, etc.). Post-update and general CO-specs are used to check that functionality *does* change, but only in expected ways. For example, new-def-shadows in the *bug-fix* update checks that, following the update, new key-value bindings properly overwrite old bindings (which was not true in the old version).

We wrote specifications to be as general as possible. For example, add-mem, on the second line of the table in Figure 4, checks that after an element is added, it is reported as present after an arbitrary sequence of function calls that does not include delete(). The code for our synthetic examples and their associated CO-specs is available on-line.[3]

[3] http://www.cs.umd.edu/projects/PL/dsu/data/dsumerge-examples.tar.gz

Redis. Figure 5 lists the updates and CO-specs for Redis. Four of the six updates required writing state transformers, often just to initialize added fields but sometimes to perform more complex transformation, e.g., the update to 1.3.9 required some reorganization of data structures storing the main database.

We found that across these updates, there were four different kinds of behavioral changes, each of which suggested a certain strategy for developing CO-specs; we employed CO-specs in each of the classes described in Section 2:

- *Unmodified behavior*: We adapted two CO-specs from our synthetic key-value store example (Figure 4), *put-get* and *new-def-shadows*, to check correct behavior of Redis' SET and GET operations over string values. As these CO-specs concern behavior that all versions of Redis should exhibit, we applied them as backward compatible CO-specs.
- *New operations*: The HASHINCRBY operation, which adds to the numeric value stored for a hash key, first appeared in version 1.3.8. We check the operation's correctness using a post-update CO-spec, *hashincrby.* The HASHINCRBY operation is supported by all later versions, and so we also developed a backward compatible *hashincrby* CO-spec for subsequent updates.
- *Modified semantics*: Before Redis version 1.3.8, a set whose last element was removed would remain in the database. We use the backward compatible CO-spec *empty-set-exists* to check this property against the patch to 1.3.7. Then for the patch to 1.3.8, which causes the server to remove a set when it becomes empty, we use a general CO-spec *empty-set-notexists* to ensure that sets are removed if they become empty after the update. Subsequent versions preserve this behavior, which we specify using a backward compatible CO-spec.
- *Conformable changes*: Redis's ZINTER operation, which computes the intersection of two sorted sets, was renamed to ZINTERSTORE in version 1.3.12. We use a conformable CO-spec, *zinter*, to specify correct behavior regardless of when an update occurs.

To make symbolic execution tractable for Redis, we had to bound the nondeterminism in our CO-specs, e.g., by limiting "arbitrary behavior" to a single operation, non-deterministically chosen from a subset of commands that relate to the specified property (rather than from the full set of Redis operations).

4.2 Effectiveness

In most cases, checking CO-specs validated the correctness of our dynamic patches. In some cases the checking found bugs. For example, in the state transformer for the multiset-to-set update, we inadvertently introduced a possible null pointer dereference when freeing duplicates. Verification with Thor discovered this problem. For Redis, we experimented with omitting state transformation code or using code with a simple mistake in it. In all cases, checking our specifications with Otter uncovered the mistakes.

Figures 4 and 5 show the running times for each of the update/CO-spec/tool combinations, listed under the **mrg** heading. As a baseline, we also list the

		Otter time (s)		
	Specification	**old**	**new**	**mrg**
→1.3.7	put-get^b	9.76	9.52	24.99
	new-def-shadows^b	2.19	2.19	3.97
	empty-set-exists^b	9.95	9.92	29.15
→1.3.8*	put-get^b	9.20	9.58	28.53
	new-def-shadows^b	2.17	2.27	4.14
	hashincrby^p		3.02	14.81
	empty-set-notexists^g			27.58
→1.3.9*	put-get^b	9.14	9.31	48.08
	new-def-shadows^b	2.27	2.66	5.46
	hashincrby^b	14.23	14.83	77.14
	empty-set-notexists^b	10.56	11.13	62.88
→1.3.10	put-get^b	9.22	10.05	27.37
	new-def-shadows^b	2.70	2.69	4.79
	hashincrby^b	14.86	15.26	46.74
	empty-set-notexists^b	11.14	11.36	35.01
→1.3.11*	put-get^b	9.85	10.04	50.73
	new-def-shadows^b	2.69	2.77	6.30
	hashincrby^b	15.19	15.51	77.80
	empty-set-notexists^b	11.33	11.57	72.40
→1.3.12*	put-get^b	10.32	9.72	49.23
	new-def-shadows^b	2.85	2.92	6.27
	hashincrby^b	15.20	14.79	77.27
	empty-set-notexists^b	11.58	11.67	72.16
	zinter^c	60.30	59.73	294.05

b – backward compatible p – post update
c – conformable g – general $*$ – xform

Fig. 5. Otter checking times for Redis

running times for the backward-compatible specifications on both individual program versions, and for post-update specifications on the new version—this lets us compare the relative slowdown incurred by reasoning about updates.

Otter. We performed experiments with Otter on a machine with a dual-core Pentium-D 3.6GHz processor and 2GB of memory. The running times range from seconds to a few minutes, depending on the complexity of the specification and the program. For example, the CO-specs for the multiset-to-set example were expensive to symbolically execute because each set insertion checks for duplicates, which induces many branches when symbolic values are involved.

We also see that, across the synthetic examples and Redis, it takes four times longer to analyze merged programs versus individual versions on average, and 6.4 times longer in the worst case. We investigated the source of the slowdown, and found it was due to the extra time required to model update points and state transformers, which is fundamental to verifying updating programs, rather than an artifact of our merging strategy. In particular, Otter runs on the merged versions, so it must explore additional program paths to model each possible update timing; on average, CO-specs reached 3.7 update points during execution and, loosely speaking, each update point could induce another full exploration through the set of non-updating program paths. State transformation is also executed following updates, so the expense of symbolically executing the transformer is multiplied by the number of times an update point is reached. Nevertheless, despite this slowdown, total checking time was rarely an impediment to checking useful properties.

Thor. We ran Thor on a 2.8GHz Intel Core 2 Duo with 4GB of memory. The average slowdown was 3.9 times, and ranged from 1.5 times to 8.3 times. Much of the slowdown derived from per-update-point analysis of the state transformation function; tools that compute procedure summaries or otherwise support modular verification would likely do better. Thor could not verify all our examples, owing to complex state transformation code and CO-specs that specify very precise properties. For example, for the multiset-to-set example, Thor was able to prove that the state transformer preserves list membership (used to verify *mem-mem*), but not that it leaves at most one copy of any element in the list (needed for *add-add-del-set*).

The CO-specs we considered lie at the boundary of what is possible for current verification technology. To verify all our examples requires a robust treatment of pointer manipulation, integer arithmetic, and reasoning about collections. We are not aware of any tools that currently offer such a combination. However, we hope that the demonstrated utility of such specifications will help inspire further research in this area.

5 Related work

This paper presents the first approach for automatically verifying the correctness of dynamic software updates. As mentioned in the introduction, prior automated analyses focus on safety properties like type safety [22], rather than correctness. As described in Section 2, our notion of client-oriented specifications captures and extends prior notions of update correctness.

Our verification methodology generalizes our prior work [10, 11] on systematically *testing* dynamic software updates. Given tests that pass for both the old and new versions, the tool tests every possible updating execution. This approach only supported backward-compatible properties and does not extend to general properties (e.g., with non-deterministically chosen operations or values).

The merging transformation proposed in this paper was inspired by KISS [19], a tool that transforms multi-threaded programs into single-threaded programs that fix the timing of context switches. This allows them to be analyzed by non–thread-aware tools, just as our merging transformation makes dynamic patches palatable to analysis tools that are not DSU-aware.

An alternative technique for verifying dynamic updates, explored by Charlton et al. [6], uses a Hoare logic to prove that programs and updates satisfy their specifications, expressed as pre/post-conditions. We find CO-specs preferable to pre/post-conditions because they require less manual effort to verify, and because they naturally express rich properties that span multiple server commands.

6 Summary

We have presented the first system for automatically verifying dynamic-software-update (DSU) correctness. We introduced *client-oriented specifications* as a way to specify update correctness and identified three common, easy-to-construct classes of DSU CO-specs. To permit verification using non-DSU-aware tools, we developed a technique where the old and new versions are *merged* into a single program and proved that it correctly models dynamic updates. We implemented merging for C and found that it enabled the analysis tool, Thor, to fully verify several CO-specs for small updates, and the symbolic executor, Otter, to check and find errors in dynamic patches to Redis, a widely-used server program.

Acknowledgements. We thank Elnatan Reisner, Matthew Parkinson, Nishant Sinha, and the anonymous reviewers for helpful comments on drafts of this paper. This research was supported by the partnership between UMIACS and the Laboratory for Telecommunications Sciences, by ONR grant N00014-09-1-0652, and NSF grants CCF-0910530, CCF-0915978 and CNS-1111698. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the ONR or NSF.

References

1. S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP*, July 2006.
2. J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent programming in ERLANG (2nd ed.).* Prentice Hall International Ltd., 1996.
3. T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, March 1993.
4. G. Bracha. Objects as software services. http://bracha.org/objectsAsSoftwareServices.pdf, Aug. 2006.
5. Cassandra API overview. http://wiki.apache.org/cassandra/API.
6. N. Charlton, B. Horsfall, and B. Reus. Formal reasoning about runtime code update. In *HOTSWUP*, 2011.
7. D. Duggan. Type-based hot swapping of running modules. In *ICFP*, 2001.
8. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.
9. D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE TSE*, 22(2), 1996.
10. C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster. Efficient Systematic Testing for Dynamically Updatable Software. In *HOTSWUP*, 2009.
11. C. M. Hayden, E. K. Smith, E. A. Hardisty, M. Hicks, and J. S. Foster. Evaluating dynamic software update safety using systematic testing, Mar. 2011.
12. M. Hicks and S. Nettles. Dynamic software updating. *ACM TOPLAS*, 27(6), 2005.
13. The K42 Project. http://www.research.ibm.com/K42/.
14. J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE TSE*, 16(11), 1990.
15. Never reboot Linux for Linux security updates : Ksplice. http://www.ksplice.com.
16. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV*, LNCS 5123, pages 428–432. Springer, 2008.
17. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, 2010.
18. I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.
19. S. Qadeer and D. Wu. KISS: Leep it simple and sequential. In *PLDI*, 2004.
20. The Redis project. http://code.google.com/p/redis/.
21. E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.

22. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. *ACM TOPLAS*, 29(4), 2007.
23. S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates for Java: A VM-centric approach. In *PLDI*, 2009.
24. C. Walton. *Abstract Machines for Dynamic Computation.* PhD thesis, University of Edinburgh, 2001. ECS-LFCS-01-425.

A Client-oriented update specifications

In Section 2 We introduced three categories of CO-spec for verifying dynamic updates: *backward compatible* specifications, *post-update* specifications, and *conformable* specifications. In this section we discuss the specification categories in more detail and show how they can be constructed mechanically given CO-specs that apply to either the old or new program version. As such, we believe that verifying a dynamic patch should add little work beyond the work already needed to verify the old and new program versions in isolation.

A.1 Backward compatible CO-specs

Most programs satisfy many of the same properties before and after a dynamic update—e.g., most of a server's behavior that the client observes is often unchanged between versions. For instance, Hayden et al. observed that OpenSSH's test suite only grew between versions—all of the old tests continued to hold as time went on [11]. This makes intuitive sense: many updates simply add new features, leaving the old features (and properties about them) unchanged, or refactor the program to improve non-functional aspects such as performance.

A *backward-compatible CO-spec* ϕ is one that holds for both the old and new versions independently. Such CO-specs are immediately usable. For example, the CO-spec in Figure 1(b) might apply to the old and new program version, and thus it immediately applies to an updating execution; assuming the update could take place during calls to get or set, we would verify that the update does not drop mappings from the store.

A.2 Post-update CO-specs

Another common category of properties consists of those that apply to the new version but not the old version. An example was given in Figure 1(c): after dropping line assume on line 23, we could apply this CO-spec directly to the new version, to verify the behavior of a newly added delete command. By adding this line, we are able to verify executions in which arbitrary operations are performed by the old version, but delete is not tested until after the update takes place.

Given a new-version CO-spec ϕ, we can mechanically transform it into a *post-update* CO-spec ϕ', as follows. We can prefix ϕ with an arbitrary sequence of calls into the old program version, ending with the assumption assume (running p_1) to ensure the new version p_1 is running when ϕ is checked. Figure 6(a) formally

```
$\mathcal{P}[\![\phi]\!] =$    while ? do
                    assume (running $p_0$);
                    if ? then $f_0(?)$ else if ? then $f_1(?)$ else ...
               ;
               assume (running $p_1$);
               $\phi$
```

(a) Post-update function $\mathcal{P}[\![\cdot]\!]$

$\mathcal{C}[\![f(v)]\!]$	$=$ if (running p_0) then $\mathcal{F}[\![f(v)]\!]$ else $f(v)$	*if* $\mathcal{F}[\![f(v)]\!]$ *defined*
$\mathcal{C}[\![f(v)]\!]$	$=$ assume (running p_1); $f(v)$	*if* $\mathcal{F}[\![f(v)]\!]$ *undefined*
$\mathcal{C}[\![$let $x = e$ in $e']\!]$	$=$ let $x = \mathcal{C}[\![e]\!]$ in $\mathcal{C}[\![e']\!]$	
$\mathcal{C}[\![$while e do $e']\!]$	$=$ while $\mathcal{C}[\![e]\!]$ do $\mathcal{C}[\![e']\!]$	
$\mathcal{C}[\![\phi']\!]$	$= \phi'$	*for all other* ϕ'

(b) Conformance function $\mathcal{C}[\![\cdot]\!]$

ϕ	$\mathcal{C}[\![\phi]\!]$
let $k = ?$ in	let $k =?$ in
let $x = ?$ in	let $x =?$ in
$set(d, k, x)$;	if (running p_0) then $set(k, v)$
	else $set(d, k, v)$;
$del(d, k)$;	assume (running p_1); $del(d, k)$;
let $x'' = get(d, k)$ in	let $x'' =$ (if (running p_0) then $get(k)$
	else $get(d, k)$) in
assert $(x'' = error)$	assert $(x'' = error)$

(c) Conforming new-version spec ϕ using $\mathcal{C}[\![\cdot]\!]$ defined in (b)

Fig. 6. Transforming new-version specifications

defines this transformation as the post-update CO-spec $\mathcal{P}[\![\phi]\!]$, where p_0 defines the functions $f_0, f_1, \ldots$. Thus, $\mathcal{P}[\![\phi]\!]$ can now be checked against an update from p_0 to p_1.

Post-update CO-specs often make sense for updates that add features or fix bugs. However, in general only CO-specs that assume the server could be in an arbitrary initial state are suitable for the post-update transformation. As a trivial example, the CO-spec assert $(get(?) = error)$ explicitly checks that our key-value store starts empty, and may not hold immediately after an update.

A.3 Conformable CO-specs

In some cases, updates change the behavior of existing features in a systematic way. For example, the Cassandra distributed database [5] added namespaces to its key-value store when moving from version 0.3 and 0.4. Thus, the new set of server functions now take a namespace identifier as an initial parameter, i.e., set(d,k,v) associates key k to value v in namespace d, and likewise get(d,k) retrieves the value associated with k in namespace d. After making this change, the developer adapts the existing single-version specifications for the old version to be compatible with the new version. For example, the specification in Figure 1(b) would be adjusted so that calls to get and set are made using some default namespace identifier.

To perform this update dynamically, the developer must write a patch π whose state transformation expression e adjusts the key-value store to be compatible with the new code—e.g., any existing key-value pairs already in the server heap could be placed in a default namespace. A reasonable choice is to have e add a default namespace d to each existing key-value pair. To test that this update provides reasonable continuity, we can take a new-version specification ϕ that uses this default namespace and adapt it so that it starts by using the old versions of the changed functions, and then changes to the new version midstream.

We can mechanize this process as follows. We assume we are given a new specification ϕ, as well as a meta-function $\mathcal{F}[\![f(v)]\!]$ that takes a call to a new-version function and transforms it to an appropriate call to an old-version function. As this may not always be possible, $\mathcal{F}[\![\cdot]\!]$ may be partial. Then we can define the meta-function $\mathcal{C}[\![\phi]\!]$ that *conforms* ϕ as shown in Figure 6(b). For our example, the developer would define $\mathcal{F}[\![get(d,k)]\!] = get(k)$ and $\mathcal{F}[\![set(d,k,v)]\!] = set(k,v)$. Note that $\mathcal{F}[\![\cdot]\!]$ bears some resemblance to Ajmani et al.'s *future simulation objects* [1], which are bits of code added to old-version servers whose aim is to convert calls from new clients to work with the old code. We are not deploying these conformance functions on-line, but rather are using them to adjust existing specifications to check proper continuity following an update.

Now suppose that the new version also adds a new function that permits a client to delete an entry: $del(d,k)$ removes any association with k from namespace d. Since there is no analogue to del defined in the old version, there is no backward translation for calls $del(d,k)$ that could appear in new-version specifications. To see how $\mathcal{C}[\![\cdot]\!]$ works in this case, consider the example given in

Figure 6(c), which shows ϕ and $\mathcal{C}[\![\phi]\!]$ side by side. Here, $\mathcal{C}[\![\phi]\!]$ permits updates to happen up until the *del* call, at which point we assume the update has taken place. (This means that the running p_0 check that follows it will always be false.)

B Handling multiple updates

The merging transformation in Figure 3 merges a program with a single update. This transformation can be generalized to prove properties about multiple updates. To see the basic idea, consider a process $\langle p, \sigma, e \rangle$ and a sequence of two updates π_1 and π_2. We would first merge $\langle p, \sigma, e \rangle$ and π_1, producing $\langle p, \sigma, e \rangle \rhd \pi_1$. Then we would merge the result with π_2, essentially producing $(\langle p, \sigma, e \rangle \rhd \pi_1) \rhd \pi_2$. To do this properly requires some small changes to the transformation. First, we need additional bookkeeping information to be passed between iterations of the transformation, e.g., instead of just g and g' as the old and new function names, we would have g^0, g^1, g^2, etc., and likewise *uflag* becomes uflag^1, uflag^2, etc. (Interestingly, no changes are needed to translations of running p, essentially since functions that have not changed are redundantly included in the patch and distinguished by the transformation.) Second, we must change $\{\![\mathsf{update}]\!\}^p$ to be the identity, i.e., to leave the new version's update keyword in place, so that it can be used to update to the next version to be merged.

With a more general merging transformation we can prove properties about multiple updates. For backward-compatible CO-specs there is no additional work since they are the same across all versions. For post-update specifications, we could generalize the transformation in Figure 6(a) so that the assumption in the loop is assume (running $p_0 \vee \cdots \vee$ running p_{n-1}) and the assumption after the loop is assume (running p_n), where the post-update CO-spec spans versions p_0 through p_n. We can make a similar generalization of the transformation in Figure 6(c) (composing multiple conformance functions together).

C Equivalence Proof

This appendix presents a formal proof of Theorem 1 from Section 3, which states that a configuration $\langle p, \sigma, e \rangle$ updated by patch π is correct if and only if the merged configuration $\langle p, \sigma, e \rangle \rhd \pi$ is correct. Note that the definition of correctness given in the paper specifies only a single patch π to be applied. For our proof, we generalize to correctness over sequences of patches $\vec{\pi}$.

C.1 Overview

The proof is structured in three parts: First, we prove a soundness lemma showing that the merged program simulates every step of execution in the old and new programs, as well as the updating step from old to new. Second, we prove a completeness lemma showing that every execution in the merged program corresponds to an execution in the original program, the new program, or the updated program. Finally, we use these results to prove the main equivalence result.

$$
\begin{aligned}
\llbracket x \rrbracket^{p,\pi} &\triangleq x \\
\llbracket a \rrbracket^{p,\pi} &\triangleq a \\
\llbracket g \rrbracket^{p,\pi} &\triangleq \begin{cases} g_{ptr} & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases} \\
\llbracket i \rrbracket^{p,\pi} &\triangleq i \\
\llbracket (v_1, v_2) \rrbracket^{p,\pi} &\triangleq (\llbracket v_1 \rrbracket^{p,\pi}, \llbracket v_2 \rrbracket^{p,\pi}) \\
\llbracket () \rrbracket^{p,\pi} &\triangleq () \\
\llbracket v_1 \; op \; v_2 \rrbracket^{p,\pi} &\triangleq \llbracket v_1 \rrbracket^{p,\pi} \; op \; \llbracket v_2 \rrbracket^{p,\pi} \\
\llbracket v_1(v_2) \rrbracket^{p,\pi} &\triangleq \llbracket v_1 \rrbracket^{p,\pi}(\llbracket v_2 \rrbracket^{p,\pi}) \\
\llbracket ? \rrbracket^{p,\pi} &\triangleq ? \\
\llbracket v_1 := v_2 \rrbracket^{p,\pi} &\triangleq \llbracket v_1 \rrbracket^{p,\pi} := \llbracket v_2 \rrbracket^{p,\pi} \\
\llbracket !v \rrbracket^{p,\pi} &\triangleq !\llbracket v \rrbracket^{p,\pi} \\
\llbracket \mathsf{ref}\ v \rrbracket^{p,\pi} &\triangleq \mathsf{ref}\ \llbracket v \rrbracket^{p,\pi} \\
\llbracket \mathsf{if}\ v\ e_1\ e_2 \rrbracket^{p,\pi} &\triangleq \mathsf{if}\ \llbracket v \rrbracket^{p,\pi}\ \llbracket e_1 \rrbracket^{p,\pi}\ \llbracket e_2 \rrbracket^{p,\pi} \\
\llbracket \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rrbracket^{p,\pi} &\triangleq \mathsf{let}\ x = \llbracket e_2 \rrbracket^{p,\pi}\ \mathsf{in}\ \llbracket e_2 \rrbracket^{p,\pi} \\
\llbracket \mathsf{while}\ e_1\ \mathsf{do}\ e_2 \rrbracket^{p,\pi} &\triangleq \mathsf{while}\ \llbracket e_1 \rrbracket^{p,\pi}\ \mathsf{do}\ \llbracket e_2 \rrbracket^{p,\pi} \\
\llbracket \mathsf{update} \rrbracket^{p,(p_\pi,e_\pi)} &\triangleq \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} \\
&\qquad \mathsf{if}\ z\ 0\ (\mathit{uflag} := ?; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!| e_\pi |\!\}^{p'}; 1)\ 0) \\
\llbracket \mathsf{assume}\ v \rrbracket^{p,\pi} &\triangleq \mathsf{assume}\ \llbracket v \rrbracket^{p,\pi} \\
\llbracket \mathsf{assert}\ v \rrbracket^{p,\pi} &\triangleq \mathsf{assert}\ \llbracket v \rrbracket^{p,\pi} \\
\llbracket \mathsf{running}\ p'' \rrbracket^{p,(p_\pi,e_\pi)} &\triangleq \begin{cases} \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ z = 0 & \text{if } p'' = p \\ \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ z \neq 0 & \text{if } p'' = p_\pi \\ \mathsf{let}\ z = 0\ \mathsf{in}\ z & \text{otherwise} \end{cases} \\
\llbracket \mathsf{error} \rrbracket^{p,\pi} &\triangleq \mathsf{error}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket p, (g, \lambda y.e) \rrbracket^{p,\pi} &\triangleq \llbracket p \rrbracket^{p,\pi}, \\
&\quad (g, \lambda y.\llbracket e \rrbracket^{p,\pi}), \\
&\quad (g_{ptr}, \lambda y.\mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ g'(y)\ g(y)) \\
\llbracket \cdot \rrbracket^{p,\pi} &\triangleq (\cdot, (\mathit{isupd}, \lambda y.\mathsf{let}\ z = !\mathit{uflag}\ \mathsf{in}\ z > 0))
\end{aligned}
$$

Fig. 7. Merging old version code.

Before we present these lemmas, we need a little notation. We define three merging transformations—for old-version, new-version, and combined-version code. The first two complete the presentation of merging given in Figure 3. Figure 7 fully defines the transformation $\llbracket \cdot \rrbracket^{p,\pi}$ which applies to old-version code, and Figure 8 fully defines $\{\!| \cdot |\!\}^{p}$, which is used with new-version code. The highlights of these transformations were explained in Section 3.3. For technical

$$
\begin{aligned}
\{\!|x|\!\}^p &\triangleq x \\
\{\!|a|\!\}^p &\triangleq a \\
\{\!|g|\!\}^p &\triangleq \begin{cases} g' & \text{if } p(g) = \lambda x.e \\ g & \text{otherwise} \end{cases} \\
\{\!|i|\!\}^p &\triangleq i \\
\{\!|(v_1, v_2)|\!\}^p &\triangleq (\{\!|v_1|\!\}^p, \{\!|v_2|\!\}^p) \\
\{\!|()|\!\}^p &\triangleq () \\
\{\!|v_1 \ op \ v_2|\!\}^p &\triangleq \{\!|v_1|\!\}^p \ op \ \{\!|v_2|\!\}^p \\
\{\!|v_1(v_2)|\!\}^p &\triangleq \{\!|v_1|\!\}^p(\{\!|v_2|\!\}^p) \\
\{\!|?|\!\}^p &\triangleq \ ? \\
\{\!|v_1 := v_2|\!\}^p &\triangleq \{\!|v_1|\!\}^p := \{\!|v_2|\!\}^p \\
\{\!|!v|\!\}^p &\triangleq \ !\{\!|v|\!\}^p \\
\{\!|\mathsf{ref}\ v|\!\}^p &\triangleq \mathsf{ref}\ \{\!|v|\!\}^p \\
\{\!|\mathsf{if}\ v\ e_1\ e_2|\!\}^p &\triangleq \mathsf{if}\ \{\!|v|\!\}^p\ \{\!|e_1|\!\}^p\ \{\!|e_2|\!\}^p \\
\{\!|\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2|\!\}^p &\triangleq \mathsf{let}\ x = \{\!|e_1|\!\}^p\ \mathsf{in}\ \{\!|e_2|\!\}^p \\
\{\!|\mathsf{while}\ e_1\ \mathsf{do}\ e_2|\!\}^p &\triangleq \mathsf{while}\ \{\!|e_1|\!\}^p\ \mathsf{do}\ \{\!|e_2|\!\}^p \\
\{\!|\mathsf{update}|\!\}^p &\triangleq \mathsf{let}\ z = 0\ \mathsf{in}\ z \\
\{\!|\mathsf{assume}\ v|\!\}^p &\triangleq \mathsf{assume}\ \{\!|v|\!\}^p \\
\{\!|\mathsf{assert}\ v|\!\}^p &\triangleq \mathsf{assert}\ \{\!|v|\!\}^p \\
\{\!|\mathsf{running}\ p''|\!\}^p &\triangleq \begin{cases} \mathsf{let}\ z = 1\ \mathsf{in}\ z & \text{if } p = p'' \\ \mathsf{let}\ z = 0\ \mathsf{in}\ z & \text{otherwise} \end{cases} \\
\{\!|\mathsf{error}|\!\}^p &\triangleq \mathsf{error}
\end{aligned}
$$

$$
\begin{aligned}
\{\!|p, (g, \lambda y.e)|\!\}^p &\triangleq \{\!|p|\!\}^p, (g', \lambda y.\{\!|e|\!\}^p) \\
\{\!| \cdot |\!\}^p &\triangleq \cdot
\end{aligned}
$$

Fig. 8. Merging new version code.

reasons, we modify the transformations slightly so that non-values (e.g., update) map to non-values (e.g., $\mathsf{let}\ z = 0\ \mathsf{in}\ z$ instead of 0). The third transformation $(\!| \cdot |\!)^{p,p_\pi}$ combines $[\![\cdot]\!]^{p,\pi}$ and $\{\!| \cdot |\!\}^p$ and returns a *set* of expressions as a result. For example, it translates function pointers g to $\{g', g_{ptr}\}$. The $(\!| \cdot |\!)^{\cdot,\cdot}$ transformation is needed because after the simulated update takes place, function pointers f may either bind to old/new versions f_{ptr} or to new versions f'.

$$
\begin{aligned}
\llparenthesis x \rrparenthesis^{p,p_\pi} &\triangleq \{x\}\\
\llparenthesis a \rrparenthesis^{p,p_\pi} &\triangleq \{a\}\\
\llparenthesis g \rrparenthesis^{p,p_\pi} &\triangleq \begin{cases} \{g', g_{ptr}\} & \text{if } p(g) = \lambda x.e\\ \{g\} & \text{otherwise} \end{cases}\\
\llparenthesis i \rrparenthesis^{p,p_\pi} &\triangleq \{i\}\\
\llparenthesis (v_1, v_2) \rrparenthesis^{p,p_\pi} &\triangleq \{(v_1', v_2') \mid v_1' \in \llparenthesis v_1 \rrparenthesis^{p,p_\pi} \wedge v_2' \in \llparenthesis v_2 \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis () \rrparenthesis^{p,p_\pi} &\triangleq \{()\}\\
\llparenthesis v_1 \ \mathit{op} \ v_2 \rrparenthesis^{p,p_\pi} &\triangleq \{v_1' \ \mathit{op} \ v_2' \mid v_1' \in \llparenthesis v_1 \rrparenthesis^{p,p_\pi} \wedge v_2' \in \llparenthesis v_2 \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis v_1(v_2) \rrparenthesis^{p,p_\pi} &\triangleq \{v_1'(v_2') \mid v_1' \in \llparenthesis v_1 \rrparenthesis^{p,p_\pi} \wedge v_2' \in \llparenthesis v_2 \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis ? \rrparenthesis^{p,p_\pi} &\triangleq \{?\}\\
\llparenthesis v_1 := v_2 \rrparenthesis^{p,p_\pi} &\triangleq \{v_1' := v_2' \mid v_1' \in \llparenthesis v_1 \rrparenthesis^{p,p_\pi} \wedge v_2' \in \llparenthesis v_2 \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis !v \rrparenthesis^{p,p_\pi} &\triangleq \{!v' \mid v' \in \llparenthesis v \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis \mathsf{ref}\ v \rrparenthesis^{p,p_\pi} &\triangleq \{\mathsf{ref}\ v' \mid v' \in \llparenthesis v \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis \mathsf{if}\ v\ e_1\ e_2 \rrparenthesis^{p,p_\pi} &\triangleq \{\mathsf{if}\ v'\ e_1'\ e_2' \mid v \in \llparenthesis v \rrparenthesis^{p,p_\pi} \wedge e_1' \in \llparenthesis e_1 \rrparenthesis^{p,p_\pi} \wedge e_2' \in \llparenthesis e_2 \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rrparenthesis^{p,p_\pi} &\triangleq \{\mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2' \mid e_1' \in \llparenthesis e_1 \rrparenthesis^{p,p_\pi} \wedge e_2' \in \llparenthesis e_2 \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis \mathsf{while}\ e_1\ \mathsf{do}\ e_2 \rrparenthesis^{p,p_\pi} &\triangleq \{\mathsf{while}\ e_1'\ \mathsf{do}\ e_2' \mid e_1' \in \llparenthesis e_1 \rrparenthesis^{p,p_\pi} \wedge e_2' \in \llparenthesis e_2 \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis \mathsf{update} \rrparenthesis^{p,p_\pi} &\triangleq \{\mathsf{let}\ z = 0\ \mathsf{in}\ z\}\\
\llparenthesis \mathsf{assume}\ v \rrparenthesis^{p,p_\pi} &\triangleq \{\mathsf{assume}\ v' \mid v' \in \llparenthesis v \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis \mathsf{assert}\ v \rrparenthesis^{p,p_\pi} &\triangleq \{\mathsf{assert}\ v' \mid v' \in \llparenthesis v \rrparenthesis^{p,p_\pi}\}\\
\llparenthesis \mathsf{running}\ p'' \rrparenthesis^{p,p_\pi} &\triangleq \begin{cases} \{\mathsf{let}\ z = 0\ \mathsf{in}\ z, \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ z = 0\} & \text{if } p'' = p\\ \{\mathsf{let}\ z = 1\ \mathsf{in}\ z, \mathit{isupd}()\} & \text{if } p'' = p_\pi\\ \{\mathsf{let}\ z = 0\ \mathsf{in}\ z\} & \text{otherwise} \end{cases}\\
\llparenthesis \mathsf{error} \rrparenthesis^{p,p_\pi} &\triangleq \{\mathsf{error}\}
\end{aligned}
$$

Fig. 9. Merging combined version code.

Next, using these transformations on expressions, we define a transformation on configurations:

$$
\begin{aligned}
&\langle p; \sigma; e\rangle \ \triangleright \ \pi \triangleq \langle \overline{p}, \overline{\sigma}[\mathit{uflag} \mapsto i], \overline{e}\rangle\\
&\langle p; \sigma; e\rangle \ [\triangleright] \ \pi \triangleq \langle \overline{p}, \hat{\sigma}[\mathit{uflag} \mapsto j], \hat{e}\rangle\\
&\text{where} \quad \overline{p} = \{\!| p_\pi |\!\}^{p_\pi}, \llbracket p \rrbracket^{p,\pi} \qquad\qquad \pi = (p_\pi, e_\pi) \qquad i \leq 0\\
&\qquad\qquad \overline{e} = \llbracket e \rrbracket^{p,\pi} \qquad\qquad\qquad\quad \hat{e} = \llparenthesis e \rrparenthesis^{p,p_\pi} \qquad j > 0\\
&\qquad\qquad \overline{\sigma} = \{l \mapsto \llbracket v \rrbracket^{p,\pi} \mid \sigma(l) = v\}\\
&\qquad\qquad \hat{\sigma} = \{\sigma' \mid \mathit{dom}(\sigma') = \mathit{dom}(\sigma) \wedge \forall l \in \mathit{dom}(\sigma).\ \sigma'(l) \in \llparenthesis \sigma(l) \rrparenthesis^{p,p_\pi}\}
\end{aligned}
$$

The $(\cdot \ \triangleright \ \cdot)$ and $(\cdot \ [\triangleright] \ \cdot)$ transformations simulate the behavior of the program before and after the update occurs respectively. Note that both transformations describe sets of configurations: $(\cdot \rhd \cdot)$ contains all configurations of the specified form where *uflag* is bound in the heap to an integer $i \leq 0$ while $(\cdot\ [\rhd]\ \cdot)$ is a set due to the use of $(\!|\ \cdot\ |\!)^{p,p_\pi}$. These sets are needed to set up the simulations between executions in the old, new, and transformed programs. To streamline the presentation we will occasionally abuse notation slightly, lifting various notions from elements to sets in the obvious way. For example, we write $\langle p;\sigma;e\rangle \rhd \pi \leadsto \langle p;\sigma';e'\rangle\ [\rhd]\ \pi$ to indicate that every configuration in $\langle p;\sigma;e\rangle \rhd \pi$ steps to a configuration in $\langle p;\sigma';e'\rangle\ [\rhd]\ \pi$.

The first lemma states that any execution in an untransformed program is matched by an execution in the transformed program.

Lemma 1 (Soundness) *For all* $p, p', \sigma, \sigma', e, e', \vec{\nu}, \pi$ *with* $\pi = (p_\pi, e_\pi)$ *we have* $\langle p;\sigma;e\rangle \overset{\vec{\nu}}{\leadsto}{}^* \langle p';\sigma';e'\rangle$ *implies*

1. *if* $\vec{\nu} = \epsilon$ *then* $p' = p$ *and* $\langle p;\sigma;e\rangle \rhd \pi \leadsto^* \langle p;\sigma';e'\rangle \rhd \pi$
2. *if* $\vec{\nu} = \pi$ *then* $p' = p_\pi$ *and* $\langle p;\sigma;e\rangle \rhd \pi \leadsto^* \langle p;\sigma';e'\rangle\ [\rhd]\ \pi$.

The second lemma states that for any execution trace of the transformed program, there is a corresponding trace of the untransformed program. However, the transformed program may need to execute a little more to match up with an untransformed state.

Lemma 2 (Completeness) *For all* $p, p', \sigma, \sigma', e, e', \pi$ *such that* $\pi = (p_\pi, e_\pi)$, *if* $\langle p;\sigma;e\rangle \rhd \pi \leadsto^* \langle p';\sigma';e'\rangle$ *then there exist* σ'' *and* e'' *such that*

– $\langle p';\sigma';e'\rangle \leadsto^* \langle p;\sigma'';e''\rangle \rhd \pi$ *and* $\langle p;\sigma;e\rangle \leadsto^* \langle p;\sigma'',e''\rangle$ *; or*
– $\langle p';\sigma';e'\rangle \leadsto^* \langle p;\sigma'';e''\rangle\ [\rhd]\ \pi$ *and* $\langle p;\sigma;e\rangle \overset{\pi}{\leadsto}{}^* \langle p_\pi;\sigma'',e''\rangle$ *; or*

Using these lemmas, we prove the main result:

Proof (of Theorem 1). Recall the statement of the theorem:

> For all p, σ, e, π with $\pi = (p_\pi, e_\pi)$ and $dom(p_\pi) \supseteq dom(p)$ we have $\models \langle p;\sigma;e\rangle, \pi$ *if and only if* $\models \langle p, \sigma, e\rangle \rhd \pi$.

We prove each direction separately.

($\Leftarrow$) Let $\langle p;\sigma;e\rangle \rhd \pi \leadsto^* \langle p';\sigma';e'\rangle$ be an execution of the transformed program. By Lemma 2 there exists a σ'' and e'' such that either:
  - $\langle p';\sigma';e'\rangle \leadsto^* \langle p;\sigma'';e''\rangle \rhd \pi$ and $\langle p;\sigma;e\rangle \leadsto^* \langle p;\sigma'';e''\rangle$. By assumption, we have $\models \langle p;\sigma;e\rangle, \pi$ and hence e'' is not error. Using Lemma 8 we also have that e' is not error.
  - $\langle p';\sigma';e'\rangle \leadsto^* \langle p;\sigma'';e''\rangle\ [\rhd]\ \pi$ and $\langle p;\sigma;e\rangle \overset{\pi}{\leadsto}{}^* \langle p';\sigma'';e''\rangle$. The result follows by a similar argument as the previous case.

($\Rightarrow$) Let $\langle p;\sigma;e\rangle \overset{\vec{\nu}}{\leadsto}{}^* \langle p';\sigma';e'\rangle$ be an execution. By Lemma 1 we have:
  - $\vec{\nu} = \epsilon$ implies $p' = p$ and $\langle p;\sigma;e\rangle \rhd \pi \leadsto^* \langle p;\sigma';e'\rangle \rhd \pi$. By assumption, we have $\models \langle p;\sigma;e\rangle, \pi$ and hence $[\![e']\!]^{p,\pi}$ is not error. Using Lemma 8 we also have that e' is not error.
  - $\vec{\nu} = \pi$ implies $p' = p_\pi$ and $\langle p;\sigma;e\rangle \rhd \pi \leadsto^* \langle p;\sigma';e'\rangle\ [\rhd]\ \pi$. The result follows by a similar argument as the previous case. □

C.2 Soundness Lemmas

The main soundness lemma follows from the three lemmas proved in this section. The first shows that the simulation between the original and transformed programs holds before to an update when taking a single step.

Lemma 3 *For all* $p, \sigma, \sigma', e, e', \pi$*, if* $\langle p; \sigma; e\rangle \rightsquigarrow \langle p; \sigma'; e'\rangle$ *then* $\langle p; \sigma; e\rangle \rhd \pi \rightsquigarrow^{+} \langle p; \sigma'; e'\rangle \rhd \pi$*.*

Proof. Let $(p_\pi, e_\pi) = \pi$ and define $\overline{p}$, $\overline{\sigma}$, and $\overline{\sigma}'$ as follows:

$$\begin{aligned} \overline{p} &= \{\!|p_\pi|\!\}^p, \llbracket p\rrbracket^{p,\pi} \\ \overline{\sigma} &= \{l \mapsto \llbracket v\rrbracket^{p,\pi} \mid \sigma(l) = v\}[\mathit{uflag} \mapsto i] \\ \overline{\sigma}' &= \{l \mapsto \llbracket v\rrbracket^{p,\pi} \mid \sigma'(l) = v\}[\mathit{uflag} \mapsto i'] \end{aligned}$$

where $i \leq 0$ and $i' \leq 0$.

The proof is by induction on $\langle p; \sigma; e\rangle \rightsquigarrow \langle p; \sigma'; e'\rangle$. Most cases are straightforward calculations using the definition of the transformation. We show just a few of the most interesting cases.

Case $\langle p; \sigma; v_1\ \mathit{op}\ v_2\rangle \rightsquigarrow \langle p; \sigma; v'\rangle$ where $v' = \llbracket \mathit{op}\rrbracket(v_1, v_2)$:
For this case, we must assume that $\llbracket \mathit{op}\rrbracket(v_1, v_2) = v'$ implies $\llbracket \mathit{op}\rrbracket(\llbracket v_1\rrbracket^{p,\pi}, \llbracket v_2\rrbracket^{p,\pi}) = \llbracket v'\rrbracket^{p,\pi}$. This rules out operators such as $<$ on function pointers (which makes intuitive sense, because relative ordering on pointers will not be preseved by the transformation in general). We calculate as follows,

$$\begin{aligned} \langle p; \sigma; v_1\ \mathit{op}\ v_2\rangle \rhd \pi &= \langle \overline{p}; \overline{\sigma}; \llbracket v_1\ \mathit{op}\ v_2\rrbracket^{p,\pi}\rangle \\ &= \langle \overline{p}; \overline{\sigma}; \llbracket v_1\rrbracket^{p,\pi}\ \mathit{op}\ \llbracket v_2\rrbracket^{p,\pi}\rangle \\ &\rightsquigarrow \langle \overline{p}; \overline{\sigma}; \llbracket \mathit{op}\rrbracket(\llbracket v_1\rrbracket^{p,\pi}, \llbracket v_2\rrbracket^{p,\pi})\rangle \\ &= \langle p; \sigma; \llbracket v'\rrbracket^{p,\pi}\rangle \rhd \pi && \text{by assumption} \\ &= \langle p; \sigma; v'\rangle \rhd \pi \end{aligned}$$

and obtain the required result.
Case: $\langle p; \sigma; \mathsf{update}\rangle \rightsquigarrow \langle p; \sigma; 0\rangle$
We calculate as follows,

$$\begin{aligned} & \langle p; \sigma; \mathsf{update}\rangle \rhd \pi \\ = \;& \langle \overline{p}; \overline{\sigma}; \llbracket \mathsf{update}\rrbracket^{p,\pi}\rangle \\ = \;& \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} \\ & \qquad \mathsf{if}\ z\ 0\ (\mathit{uflag} :=?; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^p; 1)\ 0)\rangle \\ \rightsquigarrow \;& \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = \mathsf{let}\ z = !\mathit{uflag}\ \mathsf{in}\ z > 0\ \mathsf{in} \\ & \qquad \mathsf{if}\ z\ 0\ (\mathit{uflag} :=?; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^p; 1)\ 0)\rangle \\ \rightsquigarrow^{+} \;& \langle \overline{p}; \overline{\sigma}; (\mathit{uflag} :=?; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^p; 1)\ 0)\rangle && \text{as } \overline{\sigma}(\mathit{uflag}) = i \leq 0 \\ \rightsquigarrow^{+} \;& \langle \overline{p}; \overline{\sigma}[\mathit{uflag} \mapsto i']; 0\rangle && \text{where } i' \leq 0 \\ = \;& \langle \overline{p}; \overline{\sigma}[\mathit{uflag} \mapsto i']; \llbracket 0\rrbracket^{p,\pi}\rangle \\ = \;& \langle p; \sigma; 0\rangle \rhd \pi \end{aligned}$$

and obtain the required result.

Case: $\langle p; \sigma; \mathsf{update}\rangle \overset{\pi}{\leadsto} \langle p'; \sigma; (e; 1)\rangle$ where $\pi = (p', e)$

Can't happen, as $\vec{\nu} = \epsilon$ by assumption.

Case: $\langle p; \sigma; \mathsf{running}\ p\rangle \leadsto \langle p; \sigma; 1\rangle$

We calculate as follows,

$$\begin{array}{rcll} \langle p; \sigma; \mathsf{running}\ p\rangle \ \rhd\ \pi & = & \langle \overline{p}; \overline{\sigma}; [\![\mathsf{running}\ p]\!]^{p,\pi}\rangle & \\ & = & \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ z = 0\rangle & \\ & \leadsto & \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = \mathsf{let}\ z = !\mathit{uflag}\ \mathsf{in}\ z > 0\ \mathsf{in}\ z = 0\rangle & \\ & \leadsto^{+} & \langle \overline{p}; \overline{\sigma}; 1\rangle & \text{as } \overline{\sigma}(\mathit{uflag}) = i \leq 0 \\ & = & \langle \overline{p}; \overline{\sigma}; [\![1]\!]^{p,\pi}\rangle & \\ & = & \langle p; \sigma; 1\rangle \ \rhd\ \pi & \end{array}$$

and obtain the required result.

The next lemma proves the simulation is also preserved by updates.

Lemma 4 *For all* $p, p', \sigma, \sigma', e, e', \pi$ *with* $\pi = (p_\pi, e_\pi)$ *we have* $\langle p; \sigma; e\rangle \overset{\pi}{\leadsto} \langle p'; \sigma'; e'\rangle$ *implies* $p' = p_\pi$ *and* $\langle p; \sigma; e\rangle \ \rhd\ \pi \leadsto^{+} \langle p; \sigma'; e'\rangle\ [\rhd]\ \pi$*.*

Proof. Let $(p_\pi, e_\pi) = \pi$ and define $\overline{p}$, $\overline{\sigma}$, and $\hat{\sigma}'$ as follows.

$$\begin{array}{rcl} \overline{p} & = & \{\!|p_\pi|\!\}^{p}, [\![p]\!]^{p,\pi} \\ \overline{\sigma} & = & \{l \mapsto [\![v]\!]^{p,\pi} \mid \sigma(l) = v\}[uflag \mapsto i] \\ \hat{\sigma}' & = & \{\sigma'[\mathit{uflag} \mapsto j] \mid \mathit{dom}(\sigma') = \mathit{dom}(\sigma) \wedge \forall l \in \mathit{dom}(\sigma).\ \sigma'(l) \in (\!|\sigma(l)|\!)^{p,p_\pi}\} \end{array}$$

where $i \leq 0$ and $j > 0$.

The proof is by induction on $\langle p; \sigma; e\rangle \overset{\pi}{\leadsto} \langle p'; \sigma'; e'\rangle$. We show just one case:

Case: $\langle p; \sigma; \mathsf{update}\rangle \overset{\pi}{\leadsto} \langle p_\pi; \sigma; (e_\pi; 1)\rangle$

We calculate as follows,

$$\begin{array}{rll} & \langle p; \sigma; \mathsf{update}\rangle \ \rhd\ \pi & \\ = & \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} & \\ & \qquad \mathsf{if}\ z\ 0\ (\mathit{uflag} := ?; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^{p}; 1)\ 0)\rangle & \\ \leadsto^{+} & \langle \overline{p}; \overline{\sigma}; \mathsf{let}\ z = 0\ \mathsf{in} & \text{as } \overline{\sigma}(\mathit{uflag}) = i \leq 0 \\ & \qquad \mathsf{if}\ z\ 0\ (\mathit{uflag} := ?; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^{p}; 1)\ 0)\rangle & \\ \leadsto^{+} & \langle \overline{p}; \overline{\sigma}[\mathit{uflag} \mapsto j]; \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ \mathsf{if}\ z\ (\{\!|e_\pi|\!\}^{p}; 1)\ 0\rangle & \text{where } j > 0 \\ \leadsto^{+} & \langle \overline{p}; \overline{\sigma}[\mathit{uflag} \mapsto j]; (\{\!|e_\pi|\!\}^{p}; 1)\rangle & \\ \in & \langle \overline{p}; \hat{\sigma}[\mathit{uflag} \mapsto j]; (\!|e_\pi; 1|\!)^{p,p_\pi}\rangle & \\ \subseteq & \langle p; \sigma; (e_\pi; 1)\rangle\ [\rhd]\ \pi & \end{array}$$

and obtain the required result.

The third lemma proves that the simulation is preserved following an update.

Lemma 5 *For all* $p, \sigma, \sigma', e, e', \pi$ *with* $\pi = (p_\pi, e_\pi)$ *we have* $\langle p_\pi; \sigma; e\rangle \leadsto \langle p_\pi; \sigma'; e'\rangle$ *implies* $\langle p; \sigma; e\rangle\ [\rhd]\ \pi \leadsto^{+} \langle p; \sigma'; e'\rangle\ [\rhd]\ \pi$*.*

Proof. Similar to the previous soundness lemmas.

C.3 Completeness Lemmas

The main completeness lemma follows from repeated applications of the next two lemmas.

Lemma 6 *For all $p, p', \sigma, \sigma', e, e', \pi$ where $\pi = (p_\pi, e_\pi)$, if*

$$\langle p; \sigma; e\rangle \;\rhd\; \pi \;\rightsquigarrow^{+}\; \langle p'; \sigma'; e'\rangle$$

and there does not exist σ_0 and e_0 such that either

$$\langle p; \sigma; e\rangle \rightsquigarrow \langle p; \sigma_0; e_0\rangle \quad \textit{and} \quad \langle p; \sigma; e\rangle \;\rhd\; \pi \rightsquigarrow^{+} \langle p; \sigma_0; e_0\rangle \;\rhd\; \pi \rightsquigarrow^{+} \langle p'; \sigma'; e'\rangle$$

or

$$\langle p; \sigma; e\rangle \overset{\pi}{\rightsquigarrow} \langle p'; \sigma_0; e_0\rangle \quad \textit{and} \quad \langle p; \sigma; e\rangle \;\rhd\; \pi \rightsquigarrow^{+} \langle p; \sigma_0; e_0\rangle \; [\rhd] \; \pi \rightsquigarrow^{+} \langle p'; \sigma'; e'\rangle$$

then there exists σ'' and e'' such that either

- *$\langle p'; \sigma''; e''\rangle = \langle p; \sigma'; e'\rangle \;\rhd\; \pi$ and $\langle p; \sigma; e\rangle \rightsquigarrow \langle p; \sigma''; e''\rangle$; or*
- *$\langle p'; \sigma''; e''\rangle \in \langle p; \sigma'; e'\rangle \; [\rhd] \; \pi$ and $\langle p; \sigma; e\rangle \overset{\pi}{\rightsquigarrow} \langle p_\pi; \sigma''; e''\rangle$; or*
- *$\langle p'; \sigma'; e'\rangle \overset{\nu}{\rightsquigarrow} \langle p'; \sigma''; e''\rangle$ for some ν.*

Intuitively, this lemma states that if the transformed program can take some number of steps, then either that state corresponds to a reachable untransformed state, or can take another step, eventually reaching a corresponding state.

The second lemma is similar, but considers post-update states:

Lemma 7 *For all $p, p', \sigma, \sigma', e, e', \pi$ where $\pi = (p_\pi, e_\pi)$, if*

$$\langle p; \sigma; e\rangle \; [\rhd] \; \pi \;\rightsquigarrow^{+}\; \langle p'; \sigma'; e'\rangle$$

and there do not exist σ_0 and e_0 such that

$$\langle p_\pi; \sigma; e\rangle \rightsquigarrow \langle p_\pi; \sigma_0; e_0\rangle \quad \textit{and} \quad \langle p; \sigma; e\rangle \; [\rhd] \; \pi \rightsquigarrow^{+} \langle p; \sigma_0; e_0\rangle \; [\rhd] \; \pi \rightsquigarrow^{+} \langle p'; \sigma'; e'\rangle$$

Then there exist σ'' and e'' such that either

- *$\langle p'; \sigma'; e'\rangle \in \langle p; \sigma''; e''\rangle \; [\rhd] \; \pi$ and $\langle p_\pi; \sigma; e\rangle \rightsquigarrow \langle p_\pi; \sigma''; e''\rangle$; or*
- *$\langle p'; \sigma'; e'\rangle \rightsquigarrow \langle p'; \sigma''; e''\rangle$.*

C.4 Auxiliary Lemmas

Lemma 8 (Error) *For all p, π, e, we have $e \neq \mathsf{error}$ if and only if:*

- *$\llbracket e \rrbracket^{p,\pi} \neq \mathsf{error}$;*
- *$\{\!| e |\!\}^{p} \neq \mathsf{error}$; and*
- *$\llparenthesis e \rrparenthesis^{p,\pi} \not\ni \mathsf{error}$.*

Lemma 9 (Non-Zero) *For all p, π, v, we have $v \neq 0$ if and only if:*

1. $\llbracket v \rrbracket^{p,\pi} \neq 0$*;*
2. $\{\!| v |\!\}^{p} \neq 0$*; and*
3. $(\!| v |\!)^{p,p_\pi} \not\ni 0$.

Lemma 10 (Substitution) *For all p, p', π, x, v, and e we have the following:*

- $\llbracket e[v/x] \rrbracket^{p,\pi} = \llbracket e \rrbracket^{p,\pi}[\llbracket v \rrbracket^{p,\pi}/x]$*;*
- $\{\!| e[v/x] |\!\}^{p} = \{\!| e |\!\}^{p}[\{\!| v |\!\}^{p}/x]$*; and*
- $(\!| e[v/x] |\!)^{p,p_\pi} = (\!| e |\!)^{p,p_\pi}[(\!| v |\!)^{p,p_\pi}/x]$.