Specifying and Verifying the Correctness of Dynamic Software Updates

Stephen Magill* smagill@cs.umd.edu
Christopher M. Hayden* hayden@cs.umd.edu
Michael Hicks* mwh@cs.umd.edu
Nate Foster† jnfoster@cs.cornell.edu
Jeffrey S. Foster* jfoster@cs.umd.edu

*University of Maryland, College Park †Cornell University

Abstract

Recently, there has been much interest in dynamic software updating (DSU) systems, which allow running programs to be patched on-the-fly to add features or fix bugs. Open-source and commercial products are now available to support dynamic updates to OS kernels, desktop applications, server programs, and embedded devices. However, despite the many recent advances in DSU mechanisms, techniques to reason that dynamic updates are correct have lagged behind, focusing largely on simple properties like type safety. In this paper, we present a more complete framework for specifying and verifying the correctness of dynamic updates. Our framework is capable of describing application-specific, behavioral notions of correctness. We use our framework to characterize several classes of program properties that arise in updatable software. To verify such properties in actual updatable programs, we develop a transformation that combines the old and new versions of a program into a single *merged program*. This merged program is provably equivalent to running the old and new programs in a DSU system, and, most importantly, the merged program can be analyzed by off-the-shelf program analysis tools. We describe an implementation of a program merger we have developed for C, and we demonstrate its utility with several case studies.

1. Introduction

Dynamic software updating (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 provides a service that dynamically updates commodity Linux kernels with security patches [5]. Unsanity's Application Enhancer updates executing MacOS applications with bugfixes [32]. Many web applications employ DSU techniques to provide 24/7 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. Taking a broad view, even iPhone and Android applications update dynamically: when a user navigates away from a running application, and the application code is upgraded before the user resumes it, the new version may restore any checkpointed state to pick up where the old version left off.

Given the increasing desire for DSU, an obvious question is: How do developers ensure that a dynamically updated program will behave correctly? Most DSU system builders have ignored this question, or focused on generic safety properties that rule out obviously wrong behaviors. For example, the commonly used *activeness* restriction [5, 8, 31] and the related *con-freeness* restriction prevent updates to running or soon-to-be-running components; when used properly, these restrictions ensure type safety [30]. While very useful for avoiding problems, type-safe executions may nevertheless be incorrect, a fact that has been observed empirically in past work [14].

A few researchers have studied dynamic update correctness in more detail. Gupta et al. [13] proposed *reachability* as a correctness criterion for dynamic updates. Under this criterion, an update is considered correct if the updated program eventually enters a state that could have been reached by running the new program from scratch. Unfortunately, reachability is a blunt instrument: while it is often what we want, it can allow updates that intuitively should be rejected and forbid updates that are reasonable. Consider the following example. In version 1.1.0, the vsftpd FTP server introduced a feature that limits the number of connections from any single host, implemented by maintaining a table of per-IP address connection counts. What should constitute correct behavior if we update a running vsftpd server with several active connections? A natural choice would be to preserve the connections; after all, a prime purpose of dynamic updating is to avoid service disruptions. But doing this violates reachability because the number of connections from a particular host could exceed the limit, and should these connections remain active indefinitely, we would never enter a reachable state of the new program. On the other hand, reachability would allow an update that simply terminates all existing connections; but if we are willing to do that, why update dynamically in the first place?

Others have proposed criteria that address some of the shortcomings of reachability (e.g., Ajmani et al. [4], see Section 7), but they ultimately do not provide what we think is most needed: means for *specifying* and *verifying* that a dynamic update is correct for a particular program. This paper attempts to fill this gap, making two main contributions.

First, we define a formal language syntax and semantics for specifying fine-grained dynamic update correctness properties. Specifications describe the interactions between clients and an updatable server using remote procedure calls (RPCs) to model communication. The novel feature of these specifications is the construct running p, which returns true when the server is running version p, and false otherwise. A specification that does not use this construct implicitly assumes that the server could be updated at any time. On the other hand, a specification that does use it explicitly assumes that certain interactions occur at one server version, while other interations occur at another version. For example, for the vsftpd update mentioned above, our specification could start out assuming the server is running version 1.0. After n connections are established, the specification assumes the server is running version 1.1.0, indicating that the update must have taken place some time prior to this point. If the desired semantics is for all connections to remain active after the update, we can stipulate that they must remain so. Or, if the desired semantics is that some or all of them are terminated, we can stipulate that condition, as well.

Variables	x, y, z			*Global Names*	f, g		
Operators		op		*Integers*	i, j	*Addresses*	a
Locations	l	::=	$a \mid g$				
Values	v	::=	$x \mid l \mid i \mid (v_1, v_2) \mid ()$				
Expressions	ϕ, e	::=	$v \mid v\ op\ v \mid f(v) \mid$				
			$? \mid v_1 := v_2 \mid !v \mid \mathsf{ref}\ v \mid$				
			$\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid$				
			$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid$				
			$\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{end} \mid$				
			$\mathsf{assume}\ v \mid \mathsf{assert}\ v \mid \mathsf{running}\ p$				

Figure 1. Core syntax.

While our framework is quite general, one of the goals of this work is to make it easy to derive update specifications from existing, single-version specifications. We identify three classes of common update specifications that can be derived in this way: *backward-compatible specifications* describe properties that are identical in the old and new versions; *post-update specifications* describe what must happen when new features are added by an update; and *conformable specifications* describe properties that are identical in the old and new versions, modulo a renaming of RPC function calls. Based on what we find in the literature [9, 14, 24–26, 31], we believe most update specifications will fall into one of these three classes. Thus, in many cases, the programmer can simply reuse existing single-version specifications. Of course, our framework also allows the programmer to describe arbitrary update specifications that do not fall into one of these categories, such as the ones for FTP example, above.

Our second contribution is a means to verify update specifications automatically. Conceptually, a dynamic update consists of a new version of the program and a function that transforms the state of the old version of the program into the form expected by the new version. To enable verifying dynamic updates using off-the-shelf tools, we develop a novel program transformation that takes an old program and an update and produces an ordinary program that simulates running the old program and updating it at any of the possible program points at which updates may occur. With a merged program in hand, we can reduce the problem of verifying an update specification for a patched program to the problem of verifying a single-version specification for an ordinary program. We formalize our transformation and prove that a property holds for the merged program if and only if it holds for the original program for any run during which an update may occur.

We have implemented a program merger for C programs and used it to validate update properties using two off-the-shelf tools: the symbolic executor Otter [29], and a combination of the verification tools THOR [22] and Interproc [20]. We ran both tools on a series of patches to two small programs that we wrote ourselves, a key-value store and a multiset, and we ran Otter on actual patches to Redis, an open-source storage server [3]. We found Otter in particular to be quite effective: it required little manual effort and readily found problems in incorrect patches we (inadvertently) wrote for the small programs, and correctly flagged a semantic change to Redis's behavior, manifesting as a violation of an invalidated property. We also successfully verified several of the small-program updates using THOR+Interproc, though doing so required more manual effort. As far as we are aware, our system represents the first automatic tool for verifying properties of dynamic updates.

2. Update correctness specifications

In this section, we formalize our notion of *update specifications*, which describe how a program should behave in the presence of dynamic updates. For purposes of this paper, we focus on dynamic updates that change a *server* program that interacts with one or more *clients*. By client, we mean any other system or entity interacting with the server, e.g., a wget process communicating with a web server, a BitTorrent server interacting with its peer, or an application interacting with an operating system.

Since we focus on updating servers, our notion of specification is from the perspective of the client. For example, we write specifications that capture intuitive properties such as "if we dynamically update the server from version 1.0 to version 2.0, the server must respond correctly to clients that remain active throughout the update," or, "after we perform the update, a client must be able to reliably use new features added in version 2.0."

Figure 1 formalizes a core expression language we use to describe both servers and clients. We will typically use e to denote server expressions, and ϕ to denote client expressions. This language is largely standard, but has a few notable aspects. Perhaps most importantly, we will model all interactions between a client and server as synchronous function calls $f(v)$ to a server function f. Our core language provides no bindings for function names—we introduce syntax for server function definitions below.

To support writing specifications, our language includes a nondeterministic expression $?$, which evaluates to a random integer; $\mathsf{assume}\ v$, which establishes an assumption that v is non-zero; and $\mathsf{assert}\ v$, which checks that v is non-zero. It also includes the construct $\mathsf{running}\ p$ to test whether server program p (defined next in Section 2.1) is the currently running version of the server.

Finally, all expressions are in administrative normal form [10] to keep the semantics simple. Thus, instead of $e_1 + e_2$ we write $\mathsf{let}\ x = e_1\ \mathsf{in}\ \mathsf{let}\ y = e_2\ \mathsf{in}\ x + y$, where $+$ is a built-in operator, written in the syntax as op. (We write $e_1; e_2$ as an abbreviation for $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ where x does not appear free in e_2.)

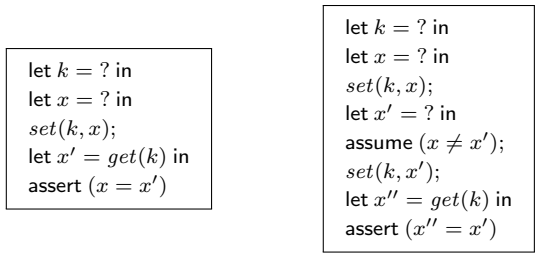
Examples As a running example we will consider a server implementing a key-value store. Some popular key-value stores include include Cassandra [2], used by Facebook, and Redis [3], used by Craiglist. These services are typically placed between an application server and a back-end database to cache the results of prior database queries.

We assume that, perhaps among other remotely-accessible functions, the key-value store server defines functions $set(k, x)$, which associates key k with value x, and $get(k)$, which returns either the associated value v or the special name $error$ otherwise. Figure 2(a) gives an example specification; function calls should be viewed as RPCs from client to server. This particular specification states that for an arbitrary pair of integers (k,x), if we associate k with x in the store via set, then a subsequent call $get(k)$ should return x. (We assume $=$ is a built-in operator that implements structural equality.) The specification in Figure 2(b) states that if we associate x with k, but then subsequently associate k with x', calling get should return the most recent value. Section 3 gives many more examples, including ones that use $\mathsf{running}\ p$.

2.1 Server and specification semantics

We now present two semantics. First, we give a semantics for server programs written in the same language as Figure 1 but extended slightly to permit dynamic updates. Second, we give a semantics for specifications (clients), which models execution of a specification against a server.

Server semantics Figure 3 gives a semantics for servers. A server program p is a mapping from function names g to functions $\lambda x.e$. Clients may call any of the functions named in the mapping. The



(a) Basic specification (b) Spec. with assumptions

Figure 2. Example specifications

Program	p	::=	$p, (g, \lambda x.e) \mid \cdot$
Expressions	e	::=	... $\mid$ update
Heaps	σ	$\in$	*Locations* $\rightharpoonup$ *Values*
Server	s	::=	(p, σ)
Patch	π	::=	(p, e)
Labels	ν	::=	$\pi \mid \epsilon$

$$
\begin{array}{lll}
\langle p;\sigma;v_1\ op\ v_2\rangle & \leadsto \langle p;\sigma;v'\rangle & v' = [\![op]\!](v_1, v_2) \\
\langle p;\sigma;\mathsf{ref}\ v\rangle & \leadsto \langle p;\sigma[a \mapsto v];a\rangle & a \notin dom(\sigma) \\
\langle p;\sigma;!l\rangle & \leadsto \langle p;\sigma;v\rangle & \sigma(l) = v \\
\langle p;\sigma;a := v\rangle & \leadsto \langle p;\sigma[a \mapsto v];v\rangle & a \in dom(\sigma) \\
\langle p;\sigma;g := v\rangle & \leadsto \langle p;\sigma[g \mapsto v];v\rangle & \\
\langle p;\sigma;?\rangle & \leadsto \langle p;\sigma;i\rangle & \text{for some } i \\
\langle p;\sigma;\mathsf{let}\ x = v\ \mathsf{in}\ e\rangle & \leadsto \langle p;\sigma;e[v\backslash x]\rangle & \\
\langle p;\sigma;f(v)\rangle & \leadsto \langle p;\sigma;e[v\backslash x]\rangle & p(f) = \lambda x.e \\
\langle p;\sigma;\mathsf{if}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle & \leadsto \langle p;\sigma;e_2\rangle & \\
\langle p;\sigma;\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle & \leadsto \langle p;\sigma;e_1\rangle & v \neq 0 \\
\langle p;\sigma;\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{end}\rangle & \leadsto & \\
\quad \langle p;\sigma;\mathsf{let}\ x = e_1\ \mathsf{in} & & \\
\quad\quad \mathsf{if}\ x\ \mathsf{then}\ e_2;\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{end} & & x \notin fv(e_1, e_2) \\
\quad\quad \mathsf{else}\ 0\rangle & & \\
\langle p;\sigma;\mathsf{update}\rangle & \leadsto \langle p;\sigma;0\rangle & \\
\langle p;\sigma;\mathsf{update}\rangle & \overset{\pi}{\leadsto} \langle p';\sigma;(e;1)\rangle & \pi = (p', e)
\end{array}
$$

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

Figure 3. Server syntax and semantics.

server program may also call any of its own functions if desired. A function body e is written in the core language extended with an additional form update to indicate a position where an update may occur (discussed more below).

A *server* is a pair containing the server program and its initial heap σ, where a heap is a partial function from locations l to values v and a location l is either a (dynamically allocated) address or a (static) global name g. We often describe heaps using set notation, writing $\{l_1 \mapsto v_i, \ldots, l_k \mapsto v_k\}$ for the heap that maps l_i to v_i for i from 1 to k and is otherwise undefined. We write $dom(\sigma)$ for the set of locations for which σ is defined and $\sigma[l \mapsto v]$ for the heap that maps l to v and otherwise behaves like σ. Note that server programs do not have closures, but since global names g are values, the language does support function pointers as in C.

Our semantics is written as a series small-step rewriting rules between *configurations* of the form $\langle p;\sigma;e\rangle$, which contain the server program p, its current heap σ, and the current expression e being evaluated. Arrows $\leadsto$ are annotated with labels ν, which are either a patch π (indicating π was dynamically applied), or the empty label ϵ; an unlabeled arrow is implicitly annotated with ϵ.

Most of the rewrite rules are straightforward. We write $e[x\backslash v]$ for the expression obtained by substituting v for every occurrence of x in e. We assume that the semantics of primitive operations op are defined by some mathematical function $[\![op]\!]$; e.g., $[\![+]\!]$ is the integer addition function. Loops are rewritten to conditionals, where guards in both cases treat non-zero values as true, and zero as false. Assignment to global names g is unusual: if the variable g does not exist then the rule creates it; conversely, addresses a must be allocated prior to assigning to them. This feature is used to allow state transformation functions, described below, to define new global variables accessible to an updated program.

The update command identifies a position in the server code at which a dynamic update may take place. Semantically, update nondeterministically 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. In the case where an update occurs, the transition is labeled with the patch π, where π is a pair (p', e) consisting of the new program code p' and an expression e that transforms the current server heap to make it compatible with the new code in p'. This expression is placed in redex position and is evaluated immediately. In practice, update would be implemented by having the run-time system check for an update, and apply it if one is available [16].

The placement of the update command has a strong influence on the semantics of updates. To permit updates only between server calls, the server developer could make update the last command of every function implementation, prior to returning, as advocated by some prior work [4, 14, 19, 25, 26]. To allow updates to take effect more readily, update could appear throughout server-side code, essentially modeling asynchronous updates, also advocated in prior work [5, 8, 12]. In practice, such an approach necessitates a timing mechanism to prevent updates from occurring at potentially-unsafe points; e.g., changes to functions f may not be permitted while f is running. To model such a timing mechanism, we could parameterize our semantics by a predicate on the patch π and the current state, and only allow update-taking transitions when the predicate is satisfied. We do not do so to keep things simple. In any case by making the update policy explicit with the update command, we can verify that such a policy does not compromise update correctness.

Specification semantics Figure 4 gives the small-step reduction rules for specifications. The main judgment is defined at the bottom of the figure as three small-step rules that relate specification configurations $\langle p;\sigma_s;\sigma_c;e\rangle$. These configurations have four components: the server p, the server heap σ_s, the client heap σ_c, and the current expression e. The first rule defines the semantics of most specification terms by appealing to the server relation $\longrightarrow$ just defined, but using the empty program $\cdot$, since specifications do not define functions. Doing so effectively rules out reductions involving function calls, and these are handled by the bottom two rules, which we will explain shortly. (Recall that we assume update does not appear in specifications ϕ. It it could, the aforementioned rule would not work as we might expect, since the update would be applied to the client heap, and an empty program.)

Figure 4 also extends the definition of $\leadsto$ to give semantics to running p, assume v, and assert v, used in specifications. (While server programs can technically use these expressions, too, we expect them to appear only in specifications.) Expression running p returns 1 if p is the server program currently running (thus we en-

$$
\begin{array}{llll}
\textit{Expressions} & \phi & ::= & \ldots \mid \mathsf{error} \\
\textit{Results} & u & ::= & v \mid \mathsf{error}
\end{array}
$$

$$
\begin{array}{lll}
\langle p;\sigma;\mathsf{running}\ p\rangle & \rightsquigarrow \langle p;\sigma;1\rangle & \\
\langle p;\sigma;\mathsf{running}\ p'\rangle & \rightsquigarrow \langle p;\sigma;0\rangle & p' \neq p \\
\langle p;\sigma;\mathsf{assume}\ v\rangle & \rightsquigarrow \langle p;\sigma;v\rangle & v \neq 0 \\
\langle p;\sigma;\mathsf{assert}\ v\rangle & \rightsquigarrow \langle p;\sigma;v\rangle & v \neq 0 \\
\langle p;\sigma;\mathsf{assert}\ 0\rangle & \rightsquigarrow \langle p;\sigma;\mathsf{error}\rangle & \\
\langle p;\sigma;\mathsf{let}\ x = \mathsf{error}\ \mathsf{in}\ e\rangle & \rightsquigarrow \langle p;\sigma;\mathsf{error}\rangle &
\end{array}
$$

$$
\frac{\langle \cdot;\sigma_c;\phi\rangle \rightsquigarrow \langle \cdot;\sigma_c';\phi'\rangle}{\langle p;\sigma_s;\sigma_c;\phi\rangle \longrightarrow \langle p;\sigma_s;\sigma_c';\phi'\rangle}
$$

$$
\frac{\langle p;\sigma_s;f(v)\rangle \overset{\vec{\pi}}{\rightsquigarrow}{}^{*} \langle p';\sigma_s';u\rangle}{\langle p;\sigma_s;\sigma_c;f(v)\rangle \xrightarrow{\vec{\pi}} \langle p';\sigma_s',\sigma_c;u\rangle}
$$

$$
\frac{\langle p;\sigma_s;\sigma_c;\phi_1\rangle \xrightarrow{\vec{\pi}} \langle p';\sigma_s';\sigma_c';\phi_1'\rangle}{\langle p;\sigma_s;\sigma_c;\mathsf{let}\ x = \phi_1\ \mathsf{in}\ \phi_2\rangle \xrightarrow{\vec{\pi}} \langle p';\sigma_s';\sigma_c';\mathsf{let}\ x = \phi_1'\ \mathsf{in}\ \phi_2\rangle}
$$

Figure 4. Specification semantics.

code a program version as the program text itself), and 0 otherwise. Expression assume v returns v if v is non-zero; execution is stuck for assume 0, identifying an irrelevant execution. Expression assert v returns v if it is non-zero, and error otherwise, which by the rule for let propagates to the top level. In short, type errors and failed assumptions result in stuck programs, while failed assertions produce an explicit error. We will discuss the reasons for this design in the next subsection.

The last two rules at the bottom of Figure 4 handle remote function calls and let bindings on the client side. The former rule pushes the call to the server, with the server's heap. The latter rule is necessary to handle function calls in specifications whose results are to let-bound variables. We write $\xrightarrow{\vec{\pi}}{}^{*}$ for the reflexive, transitive closure of the $\longrightarrow$ relation. As each step $\longrightarrow$ is potentially labeled with an update π, the transitive closure is labeled with a vector of updates $\vec{\pi}$ consisting of the sequence of non-empty labels associated with individual transitions (empty labels are discarded).

2.2 Specification satisfaction

We can now define *specification satisfaction*, written $s, \vec{\pi} \models \phi$, which holds if and only if none of the possible executions of specification ϕ on server program s that apply any subsequence of the updates $\vec{\pi}$ result in a failed assertion. Formally:

Definition 1 (Satisfaction)**.** *A server program $s = (p, \sigma)$ and a sequence of updates $\vec{\pi}$ satisfy a specification ϕ, denoted $s, \vec{\pi} \models \phi$, if and only if for all $p', \sigma_s, \sigma_c, e, \vec{\pi_0}$ such that $\vec{\pi_0}$ is a prefix of $\vec{\pi}$ it is the case that $\langle p;\sigma;\cdot;\phi\rangle \xrightarrow{\vec{\pi_0}}{}^{*} \langle p';\sigma_s;\sigma_c;e\rangle$ implies e is not* error.

Notice that specification satisfaction does not imply type-correct executions as expressions such as !1 are simply stuck. We could add rules that transition such bogus expressions to error, but we do not do so to keep things simpler; there are many well-known techniques for avoiding type errors in dynamic updates, such as the activeness or con-freeness restrictions [5, 8, 30, 31].

When the sequence of updates is empty we simply write $s \models \phi$. This states that every non-updating execution of the server s satisfies the specification ϕ, corresponding to standard notions of correctness for single-version programs. The relation $s, \pi \models \phi$ states that server s satisfies ϕ when subject to a single update π.

3. Defining update specifications

Given an old server s_0 of the form (p_0, σ_0), a new server s_1 of the form (p_1, σ_1), and a patch π of the form (p_1, e) that updates the server to p_1, a developer needs a way to check that s_0 behaves correctly when dynamically updated using π. In this section, we show that in many cases such update specifications can be derived from *existing* single-version specifications used for verification—i.e., ϕ such that either $s_0 \models \phi$ or $s_1 \models \phi$, or both. We identify three classes of update specification that can be derived systematically: *backward compatible* specifications, *post-update* specifications, and *conformable* specifications. Otherwise, *general* specifications that characterize update-specific behavior can be written by hand. Such general specifications require careful consideration, but are also the most interesting and the most rare. In short, we believe that verifying a dynamic patch should add little work beyond the work already needed to verify s_0 and s_1 in isolation.

3.1 Backward compatible specifications

Most programs satisfy many of the same properties before and after a dynamic update—i.e., most of the server's behavior that the client observes is unchanged between versions. For instance, Hayden et al. [14] observed that OpenSSH's test suite only grew between versions—all of the old tests continued to hold as time went on. 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-compatibile specification* ϕ is one that holds for both s_0 and s_1 independently, i.e., $s_0 \models \phi$ and $s_1 \models \phi$. Such specifications are immediately usable. As an example, suppose that s_0 and s_1 are different versions of a key-value store and further suppose that the specification in Figure 2(a), call it ϕ, holds for both versions. Now we want to check that ϕ is preserved when updating from s_0 to s_1. This is easy to do: we use ϕ exactly as is, but now try to verify $s_0, \pi \models \phi$. Performing this verification could reveal bugs in the state transformation expression e packaged with π. In particular, imagine that update appears in the server code at the end of every exported function; i.e., an update, if available, will be applied at the conclusion of any remote function execution. Now suppose that e is written incorrectly so that it resets the store. In this case, $s_0, \pi \not\models \phi$ because when the update is applied at the conclusion of the set operation, the result of get will fail to match since the store is now empty.

3.2 Post-update specifications

Another common category of properties consists of those that apply to the new version but not the old version, i.e., ϕ such that $s_1 \models \phi$ but $s_0 \not\models \phi$. As an example, suppose that s_0 is our key-value store and it contains a bug: set does not properly override existing associations, but leaves intact the initial association. Server s_1 fixes this bug and the developer writes the specification in Figure 2(b), call it ϕ, to verify as much; i.e., she proves the fix is correct by checking $s_1 \models \phi$. We cannot verify $s_0, \pi \models \phi$ directly since there are some executions for which this property fails to hold; e.g., those for which the update takes place after the second call to set. However, we can easily derive an update specification from ϕ that we can use to verify the update.

Observe that ϕ should hold regardless of the initial state it is executed in. Thus, we can transform ϕ into a *post-update* specification ϕ' that prefixes 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. More formally, we can define the *post-update* specification $\mathcal{P}[\![\phi]\!]$ as given in Figure 5(a) where $p_0 = (f_0, \lambda x.e_0), (f_1, \lambda x.e_1), \ldots$. Thus, $\mathcal{P}[\![\phi]\!]$ can now be checked against an update from p_0 to p_1. For our example $\mathcal{P}[\![\phi]\!]$ would produce a specification that performs an arbitrary number of *set* and *get* operations using the old code, expects the update to take place sometime during the last such operation, and then checks ϕ on the updated program.

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

3.3 Conformable specifications

In some cases, updates add new features that change the behavior of existing features, but they do it in a systematic way. To illustrate, suppose a developer writes s_1 to extend our key-value store s_0 with support for namespaces. Such a change occurred between versions 0.3 and 0.4 of the Cassandra distributed database [2]. 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 s_0 to be compatible with the new version. For example, the specification in Figure 2 would be adjusted so that calls to *get* and *set* are made using some default namespace identifier.

To update s_0 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 formalize 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 5(b). For our example, the developer would define $\mathcal{F}[\![get(k)]\!] = get(d, k)$ and $\mathcal{F}[\![set(k, v)]\!] = set(d, k, v)$. Note that $\mathcal{F}[\![\cdot]\!]$ bears some resemblance to Ajmani et al.'s *simulation objects* [4]; see Section 7.

Now suppose that s_1 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 s_0, there is no backward translation for calls $del(d, k)$ that could appear in s_1 specifications. To see how $\mathcal{C}[\![\cdot]\!]$ works in this case, consider the example given in Figure 5(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.)

3.4 General specifications

Some updates change program behavior in ways that cannot be easily tested using existing specifications. For example, suppose that server s_1 changes key-value store s_0 by limiting the number of keys stored in any given namespace. To ensure this limit is met post-update, the developer might write a state transformation function that randomly removes key-value pairs from namespaces that exceed the limit. No existing specification can accurately verify this behavior: old-version specifications assume no limit (so all *get*s following a *set* on the same key should succeed), while new-version specifications assume deterministic behavior, e.g., once the limit is reached, no new pairs are allowed, but old pairs remain available. Thus, to verify this particular update's behavior, the developer needs to write an appropriate update specification from scratch. It is very likely the developer's update specification will need to call running p to reason about when the update, with its randomized state transformer, has been executed.

$\mathcal{P}[\![\phi]\!] =$

```
while ? do
    assume (running p0);
    if ? then f0(?) else if ? then f1(?) else ...
end;
assume (running p1);
φ
```

(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*
$\mathcal{C}[\![\text{let } x = e \text{ in } e']\!]$	= let $x = \mathcal{C}[\![e]\!]$ in $\mathcal{C}[\![e']\!]$	
$\mathcal{C}[\![\text{while } e \text{ do } e' \text{ end}]\!]$	= while $\mathcal{C}[\![e]\!]$ do $\mathcal{C}[\![e']\!]$ end	
$\mathcal{C}[\![\phi']\!]$	$= \phi'$	*for all other ϕ'*

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

$\underline{\phi}$	$\underline{\mathcal{C}[\![\phi]\!]}$
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) Example conformance of spec from Fig. 2(b)

Figure 5. Transforming new-version specifications

4. Verification via program merging

We now describe a way to verify that an update specification ϕ is satisfied. While much technology exists for verifying program properties, all of it assumes a single, static program. In this section, we describe a program transformation that takes an old program (p, σ) and a patch π and yields a single *merged program* $(p, \sigma) \triangleright \pi$. The semantics of the merged program is provably equivalent to the original program with the patch applied to it. Formally, we can show the following theorem, where $[\![\phi]\!]^{p,\pi}$ is a transformed version of ϕ:

Theorem 1 (Equivalence)**.** *For all p, σ, π, ϕ such that $\pi = (p', e')$ and $dom(p') \supseteq dom(p)$ we have that $(p, \sigma), \pi \models \phi$* if and only if *$((p, \sigma) \triangleright \pi) \models [\![\phi]\!]^{p,\pi}$.*

$$
\begin{array}{rcl}
[\![p', (g, \lambda y.e)]\!]^{p,\pi} & = & [\![p']\!]^{p,\pi}, \\
& & (g, \lambda y.[\![e]\!]^{p,\pi}), \\
& & (g_{ptr}, \lambda y.\mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} \\
& & \quad \mathsf{if}\ z\ \mathsf{then}\ g'(y)\ \mathsf{else}\ g(y)) \\
[\![\cdot]\!]^{p,\pi} & = & (\cdot, (\mathit{isupd}, \lambda y.\mathsf{let}\ z = !\mathit{uflag}\ \mathsf{in} \\
& & \quad z > 0)) \\
\hline
[\![\sigma]\!]^{p,\pi} & = & \{l \mapsto [\![v]\!]^{p,\pi} \mid \sigma(l) = v\} \\
\hline
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]^{p,\pi} & = & \mathsf{let}\ x = [\![e_1]\!]^{p,\pi}\ \mathsf{in}\ [\![e_2]\!]^{p,\pi} \\
[\![\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]^{p,\pi} & = & \mathsf{if}\ [\![v]\!]^{p,\pi}\ \mathsf{then}\ [\![e_1]\!]^{p,\pi}\ \mathsf{else}\ [\![e_2]\!]^{p,\pi} \\
[\![v(v')]\!]^{p,\pi} & = & [\![v]\!]^{p,\pi}([\![v']\!]^{p,\pi}) \\
[\![\mathsf{update}]\!]^{p,(p',e)} & = & \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} \\
& & \mathsf{if}\ z\ \mathsf{then}\ 0\ \mathsf{else} \\
& & \quad \mathit{uflag} := ?; \\
& & \quad \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} \\
& & \quad \mathsf{if}\ z\ \mathsf{then}\ (\{\!|e|\!\}^{p'}; 1)\ \mathsf{else}\ 0 \\
[\![\mathsf{running}\ p]\!]^{p,\pi} & = & \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ z = 0 \\
[\![\mathsf{running}\ p']\!]^{p,(p',e)} & = & \mathit{isupd}() \\
[\![\mathsf{running}\ p'']\!]^{p,(p',e)} & = & 0 \quad p'' \neq p \wedge p'' \neq p' \\
[\![g]\!]^{p,\pi} & = & g_{ptr} \quad \text{if } p(g) = \lambda x.e \\
[\![l]\!]^{p,\pi} & = & l \quad \text{(otherwise)} \\
[\![v_1 := v_2]\!]^{p,\pi}, \text{etc.} & = & \text{transform } v_i \text{ structurally, as for if}
\end{array}
$$

Figure 6. Transforming old-program code.

$$
\begin{array}{rcl}
\{\!|p', (g, \lambda y.e)|\!\}^{p} & = & \{\!|p'|\!\}^{p}, (g', \lambda y.\{\!|e|\!\}^{p}) \\
\{\!|\cdot|\!\}^{p} & = & \cdot \\
\hline
\{\!|\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2|\!\}^{p} & = & \mathsf{let}\ x = \{\!|e_1|\!\}^{p}\ \mathsf{in}\ \{\!|e_2|\!\}^{p} \\
\{\!|\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2|\!\}^{p} & = & \mathsf{if}\ \{\!|v|\!\}^{p}\ \mathsf{then}\ \{\!|e_1|\!\}^{p}\ \mathsf{else}\ \{\!|e_2|\!\}^{p} \\
\{\!|v(v')|\!\}^{p} & = & \{\!|v|\!\}^{p}(\{\!|v'|\!\}^{p}) \\
\{\!|\mathsf{update}|\!\}^{p} & = & 0 \\
\{\!|\mathsf{running}\ p|\!\}^{p} & = & 1 \\
\{\!|\mathsf{running}\ p'|\!\}^{p} & = & 0 \quad p \neq p' \\
\{\!|g|\!\}^{p} & = & g' \quad \text{if } p(g) = \lambda x.e \\
\{\!|l|\!\}^{p} & = & l \quad \text{(otherwise)} \\
\{\!|v_1 := v_2|\!\}^{p}, \text{etc.} & = & \text{transform } v_i \text{ structurally, as for if}
\end{array}
$$

Figure 7. Transforming new-program code.

$$
\begin{array}{rcl}
(p, \sigma_s) \rhd \pi & \triangleq & (\overline{p}, \overline{\sigma_s}) \\
& & \text{where} \\
\pi & = & (p', e') \\
i & \leq & 0 \\
\overline{p} & = & \{\!|p'|\!\}^{p'}, [\![p]\!]^{p,\pi} \\
\overline{\sigma_s} & = & [\![\sigma_s]\!]^{p,\pi}[\mathit{uflag} \mapsto i]
\end{array}
$$

Figure 8. Transforming programs.

In other words, a server (p, σ) patched with π satisfies a specification ϕ if and only if the merged program $(p, \sigma) \rhd \pi$ satisfies $[\![\phi]\!]^{p,\pi}$. This result lets us use stock verification tools to check properties of programs with updates instead of having to develop new tools or extend existing ones.

For simplicity, we require that the updated program p' does not delete any functions in p. The merging strategy applies to a single program update. Our approach can be readily generalized to arbitrary sequences of updates, but we do not do so here to keep the description simpler.

The definition of $(p, \sigma) \rhd \pi$ is given in Figure 8. It makes use of functions $[\![\cdot]\!]^{\cdot}$ and $\{\!|\cdot|\!\}^{\cdot}$ that are defined in Figures 6 and 7, respectively. The transformation of specifications ϕ in the theorem also uses the $[\![\cdot]\!]^{\cdot}$ function.

The merging transformation is quite simple: It handles functions by renaming each new-version function from g to g', to distinguish the old and new versions, and changing all new-version code to call g' instead of g (see Figure 7). 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 happened (see Figure 6). The transformation introduces a global variable uflag and a function isupd to keep track of whether the update has taken place. It then rewrites all calls to g in the old version to call g_{ptr} instead.

The transformation rewrites updates `update` appearing in old-version code into expressions that, when reached, check whether uflag is positive. If yes, then the update has already taken place, so there is nothing to do. Otherwise, the transformation sets uflag to a non-deterministic value, which simulates a non-deterministic choice to apply the update or not. If uflag now has a positive value, the update is viewed as having taken place, so the transformation executes the developer-provided state transformation e, which is is also transformed according to $\{\!|\cdot|\!\}^{\cdot}$ to properly reference functions in the new program. The transformation translates version tests `running` p into calls to isupd.

We prove the equivalence theorem via bisimulation, showing that the semantics of a specification using the original program simulates the transformed specification and program, and vice versa. In the supplemental material we give the proof of this theorem and describe the key lemmas. The proofs of these lemmas will be made available in a forthcoming technical report.

5. Implementation

We have implemented a tool suite that closely follows the formal development from Sections 2 to 4. We write specifications as C program fragments that model clients that call server functions. We have implemented a program merger that combines two versions of a server written in C, a state transformer, and a client specification into a single program. We can then check and verify the merged program using off-the-shelf tools; we discuss our experience doing so in the next section.

5.1 Specifications in C

In our formal framework, client requests and server responses are modeled as direct function calls. In practice, such interactions take place via some more primitive communication mechanism. For example, a server may be structured using an event loop pattern, where there is a single loop that parses input from the network and dispatches requests to the appropriate internal functions. The responses from these internal functions are directed back to clients via the event loop code. For example, a web server event loop might read the HTTP command `GET /index.html` from a socket, parse this command, and then call `get("/index.html")`, where `get` is an internal function whose response is marshaled and sent back to the

```
1  // Key-value interface
2  int get(int k, int *v) {
3    ...
4    update();
5    return ...
6  }
7  void set(int k, int v) {
8    ...
9    update();
10 }
11
12 // Sample specifications
13 void arbitrary () {
14   int k = nondet();
15   int v = nondet();
16   if (nondet())
17     get(k,&v);
18   else
19     set(k,v);
20 }
```

```
21 void back_compat_spec() {
22   int k = nondet();
23   int v_in = nondet();
24   int v_out, found;
25   while(nondet())
26     arbitrary ();
27   set(k, v_in );
28   found = get(k,&v_out);
29   assert (found &&
30           v_out == v_in);
31 }
32 void post_update_spec() {
33   int k = nondet();
34   int v_out, found;
35   while(nondet())
36     arbitrary ();
37   assume_updated();
38   delete (k);
39   found = get(k,&v_out);
40   assert (!found);
41 }
```

Figure 9. Sample C specifications for key-value store.

client. We refer to functions like get as *interface functions* since they implement the logic of the server's external interface.

Thus, to verify a typical server using client-oriented specifications such as those in Figure 2, a verification tool would have to properly model client-server communication via sockets, interprocess communication, etc. Unfortunately, we know of no verification tools that are able to do this. Therefore, we take an approach very close to how we have modeled the situation formally: *client specifications are C code that call a server's interface functions directly*. From the point of view of a verification tool, the specification is the entry point to the program (rather than main()). We arrange this entry point to begin with any initialization code that would have run prior to executing the server's actual event loop, and then proceed to executing our client specification. We have manually developed specifications in this manner for the open-source key-value store Redis [3] and found it not too difficult.

The specifications we write in C are very close to the formal specifications shown earlier, with a few surface-level differences. In particular: we implement the update command as a call to function update(), non-deterministic integers ? as calls to nondet(), and version assumptions assume running p' as calls assume_updated() (recall that merging only considers a single update). The last is akin to directly calling the $isupd$ function introduced by the merging transformation. Unlike the formal framework, we allow client specifications to define functions for convenience.

Figure 9 shows an example of two specifications and snippets of code for a key-value server that supports get and set functions. Lines 1–10 represent the implementations of the server functions; the salient point is that a call to update() appears at the end of each, indicating that updates may take place only when server functions complete. The remainder of the figure shows the specification code. The function arbitrary (lines 13–20) is used by both specifications. It non-deterministically calls one of the two server functions with non-deterministic arguments. The first specification, back_compat_spec, performs an arbitrary sequence of server operations and then checks that an element subsequently added by set will be returned by get. This is a backward compatible specification and thus holds regardless of where the update occurs (Section 3.1). The second specification, post_update_spec, targets an update that adds a delete operation. It checks that, following the update, the delete operation performs as expected. This is an example of a post-update specification (Section 3.2).

5.2 Merging

Our program merger is implemented on top of the C Intermediate Language (CIL) [27], a C parser and intermediate representation written in Objective Caml. The merger takes two arguments, and old program and a new program, and produces a merged program whose entry point is the old program's main() function. We assume that the new program contains a distinguished function state_xform() that is meant to be called when an update takes place; this call plays the role of the expression e in a patch π.

To use the merger to perform verification, the user is responsible for setting up the old program's main() function appropriately. We do this as follows. First, we extract the initialization code in the normal main() into a separate init() function, and rename the existing main() function. Second, we put all our specification functions (e.g., like those in Figure 9) into a separate file that is considered part of the old program. Finally, for each specification, we run a script that creates a custom main() for each specification—this simply calls init() and then calls the specification function. For each custom main() function we merge the old and new program to create a viable verification target.

Our merger is very close to the formal description given earlier. One difference is that we will often uniformly rename the global variables of the program version to simplify writing state_xform(). For example, suppose a global variable curtime in the program is changed from an int to a float. Then we automatically rename this variable to be curtime_new, and the programmer can initialize it in state_xform() by the assignment curtime_new := itof(curtime).

On a related note, we must also take care in how we treat functions whose types have changed. Consider our example from Section 3.3 in which we add a namespace parameter to get and set. In the formalism, get takes a single argument, which is a key in the old version, and a namespace-and-key pair in the new version. The get_{ptr} function introduced by the transformation in Figure 6 simply passes its argument, whether key or pair, to the appropriate version. We cannot write such a function in C, however, because C functions take multiple arguments.

It turns out this is not as deep of a problem as it might seem. For functions g whose type changes, we can produce g_{ptr} whose true branch fails, i.e., contains assert 0. This is reasonable because new code will invoke the new version directly—function names g that appear in new code are transformed to g', not g_{ptr}; see Figure 7. Any call to g_{ptr} must be from old code, rather than new code, and thus the call is a legitimate type error—in the actual program, the call would pass the wrong number of arguments. This transformation simulates Stoyle et al.'s previously proposed analysis for avoiding type-incorrect updates [30], but is more precise since we use a path-sensitive, rather than simply flow-sensitive, analysis (as mentioned below).

6. Experiments

With a merged program we can check update specifications using off-the-shelf tools. This section describes our experience doing so trying several tools on a variety of updates, some of which we constructed ourselves, inspired be realistic changes, and some of which we acquired from Redis [3]. We begin by describing the tools we have tried, and what we find works well with them and what does not, and then describe our experiences with particular updates in more detail. As far as we are aware, ours is the first study of statically verifying dynamic updates.

Update	Specification	Type
key-value store — bug-fix		
	put-get-distinct*	backward compatible
	new-def-shadows*	general
	new-def-shadows* (fails)	backward compatible
key-value store — added namespaces		
	new-def-shadows-postupdate	post-update
	put-get-postupdate	post-update
	new-def-shadows-conformed	conformable
	put-get-conformed	conformable
key-value store — no duplicate keys, bad state transformer		
	put-get (fails)	backward compatible
	new-def-shadows (fails)	backward compatible
key-value store — no duplicate keys, correct		
	put-get	backward compatible
	new-def-shadows	backward compatible
multiset — change to set, correct		
	mem-mem*	backward compatible
	add-mem*	backward compatible
	add-add-del-set*	general
	add-add-del-multiset*	general
multiset — change to set, broken		
	mem-mem	backward compatible
	add-mem*	backward compatible
	add-add-del-set* (fails)	general
	add-add-del-multiset*	general

Figure 10. Summary of verified update specifications. *Indicates examples that were verified with THOR.

6.1 Tools

To verify a dynamic update translated into a merged program, a verification tool must provide at least some degree of path-sensitivity to be able to accurately model the state of $uflag$. A path-insensitive tool would incorrectly consider calls to old-version functions to be possible after an update, which would likely to lead to false alarms. In practice, we found that support for reasoning about pointer-based data structures is also important for verifying interesting properties. We have experimented with two classes of tools: *symbolic execution* tools, and *verification* tools. These two classes present a trade-off: the former are easier to use and scale better, but provide incomplete assurance, while the latter can provide complete assurance but do not scale as well and require more manual effort.

Symbolic Execution Using a symbolic executor, programmers can identify certain values as *symbolic*, meaning they represent arbitrary concrete program values. When the program uses symbolic data, the symbolic executor separately considers all feasible paths along which a possible concrete valuation of the data could lead. Symbolic execution has gained popularity in recent years, with KLEE [18] and SAGE [11] being exemplars.

We have experimented with using Otter, a symbolic executor for C developed by Reisner et al [29], to check update specifications. We find that it works quite well: checking is reasonably fast, works on non-toy programs, and requires little manual effort. We also tried using KLEE; its incomplete support for standard library code (used extensively by Redis) was a hindrance, but we doubt there are any fundamental problems. With Otter, we implement nondet() as returning a fresh symbolic value. Thus, conditioning on nondet() will cause both branches to be explored. Otter supports assume and assert directly.

While easy to use, symbolic executors provide incomplete assurance. They explore each program path individually, so they can only consider a finite number of paths. For potentially non-terminating programs, this means full verification is not possible. Nevertheless, by replacing unconstrained loop guards in specifications (e.g., Line 25 of Figure 9) with small bounds, Otter was able to find bugs in dynamic updates in relatively short order.

Verification We refer to tools that attempt to prove that properties hold for all executions of a program as verification tools. Examples of such tools include model checkers, such as Blast [15] and Spin [17], and abstract interpreters such as ASTREE [1] and Interproc [20]. We originally attempted to use Blast to perform our verification experiments. However, the heavy use of pointer-based data structures by our example programs proved problematic. This issue arises not from merged programs, but from the nature of the original programs.

We were able to overcome these problems using a version of THOR [22], a static analysis tool specialized for programs that use heap-based data structures. Because our specifications state correctness properties that go beyond standard safety properties, they pose a challenge for many verification tools (and THOR had to be modified to accommodate them). Nevertheless, tools are constantly improving, and because our verification strategy is based on program merging rather than any particular verification tool, it can take advantage of these improvements as they emerge. The update specifications we have explored may even constitute useful examples to help spur further tool development.

6.2 Small key-value store and multiset

Figure 10 summarizes a set of example patches we developed for the key-value store and multiset. We will explain the figure shortly. We used Otter to check whether these updates satisfy the specifications. Otter flags specifications that do not hold, but cannot verify the rest, since its exploration is bounded. In the cases marked with an asterisk, we were then able to use THOR and Interproc to prove that these potentially true specifications hold over all program runs.

Key-value store We worked with five versions of the key-value store, constituting four updates; each update is grouped and labeled in Figure 10, along with names of specifications we checked for that update, and the category to which they belong (e.g., backward compatible, post-update, etc.). Each individual version of the program is 40-50 lines excluding state transformation and specifications.

The first update fixes a bug where only the value from the first call to set for a particular key is stored, while values from subsequent sets for that key are ignored. We developed two specifications to check for correct new-version behavior. The first, *put-get-distinct*, is a backward-compatible specification, in which we verify a key inserted for the first time is properly mapped. The second, *new-def-shadows*, is a general specification similar to the one in Figure 2(b), except with assume_updated() in place of assume($x \neq x'$). Without one of these assumptions the specification rightly fails to verify (as indicated by the line marked "(fails)").

The next update adds namespaces for indexing into the key-value store, as described in Section 3.3. We verified post-update specifications for two properties of the new version. Because all of the old-version operations have close analogs in the new version, old-version specifications can be conformed to work at the new version by adding a default namespace. We constructed two such conformable specifications and successfully verified them.

The third update is a memory-use optimization to the key-value store that discards duplicate elements. The put-get specification checks a slightly more general version of back_compat_spec in Figure 9 and the second specification checks that old bindings are properly shadowed by new bindings. Our state transformation function iterates through the existing state and removes duplicate

elements, to establish the invariant expected by the new code. Our first attempt at writing this transformer had a bug in it which failed to remove all duplicates. Happily, Otter uncovered the bug by failing to check the specification. The next grouping in the figure shows the verification results for the corrected patch.

Multiset / Set We considered two patches, one correct and one broken, that update a container so that it provides set semantics, rather than multiset semantics. Each program version is roughly 60 lines long, excluding state transformation and specifications.

In the broken patch, the state transformer leaves the state unmodified, while in the correct patch, the state transformer correctly removes duplicates. In each case we verified two backward compatible properties: mem-mem, which requires that two tests of membership for some element, without intervening deletes, return the same value, and add-mem, which requires that an add of some element followed by a membership test for that element returns true provided there were no intervening deletes. Both of these properties passed for each patch. The last two specifications describe the precise update timing at which set semantics are first observed. These specifications state that a delete operation should provide set semantics whenever the update occurs prior to the delete. This specification could only be verified for the patch containing a state transformer that removes duplicates, as expected.

6.3 Redis

Redis [3] is an open-source server that provides access to and storage for a variety of container types including sets and hashes (the latter making it a kind of key-value store). We developed two dynamic patches for Redis, one for updating version 1.3.6 to 1.3.7, and the other for version 1.3.7 to 1.3.8, and we developed update specifications in both cases. Each version is significantly larger than the simple examples just presented, with each version roughly 12k lines of C code. We used Otter to check our specifications; we have yet to try using THOR+Interproc.

The first update did not affect the externally-observable behavior of the server, so we constructed two backward-compatible specifications. The first requires that, if a set operation that creates a mapping between a key and a value is followed by arbitrary sets and gets (where none of the sets is to the initial key), then a subsequent get of the initial key will return the initial value. The second specifies that if an element is added to a set, followed by arbitrary add and remove operations, a check for the existence of the set will succeed; the set may be empty, but it will still exist on the server.

The second update implemented several observable changes to behavior. Among other changes, the semantics of set removal changed: if a remove operation deletes the last element of a set, the set itself is deleted. We attempted to check the two backwards-compatible specifications from the previous patch. Our technique successfully found that the second spec was no longer valid due to the change in set-remove semantics. We constructed a post-update spec to check the new behavior, and a post-update spec to check the behavior of a new hash-value increment operation.

To ensure that specification checking using Otter terminated relatively quickly, we limited the use of loops of arbitrary operations to one or two kinds of operations in each case, rather than exploring branches for each of the dozens of operations that Redis supports. To get a sense of the running time of merged-program checking compared to single-version checking, we measured the running times of the three backward-compatible specifications on the original, unmerged program, with their running time on the merged program. In the former case the running times were 9.19s, 9.74s, and 9.18s (the average of 5 executions), while in the latter, the running times were 23.94s, 28.82s, and 28.60s, respectively, for each specification. Though we did not explore the reasons for the difference in depth, it is intuitively clear that for merged programs additional exploration is needed since each update point increases the number of paths to be explored.

7. Related work

This paper is distinguished from prior related work in two ways. First, our notion of update specification generalizes various correctness criteria proposed previously. Second, we are the first to implement automatic techniques for verifying update specifications.

The earliest work we are aware of that attempts to formally define a notion of correctness for dynamic updates is by Bloom and Day [6, 7]. They observed that requiring an update to preserve every observable behavior of the program being updated is too inflexible in practice (nevertheless, later work by others stipulated this strict notion of correctness [19]). They therefore relaxed this condition to allow "future-only implementations" and "invisible extensions", which are classes of programs obeying specifications similar to our post-update and backward-compatible specifications.

Gupta et al. [13] proposed a criterion called *reachability* that relaxes the one identified by Bloom and Day by allowing updates that *eventually* reach some state of the new program. They proved that reachability is undecidable in general, but showed that certain classes of updates based on a notion of "functional enhancements" always satisfy it. As discussed in Section 1, the added flexibility offered by this definition is both a blessing and a curse in that it permits intuitively useful behaviors as well as detrimental ones.

Ajmani et al. [4] studied dynamic update in a richer setting where multiple versions of the program are allowed to execute simultaneously. Every update includes single-version specifications for the old and new program, expressed in the usual way as pre-conditions and post-conditions, as well as an *update invariant*, discussed below. They also proposed *simulation objects*, which provide a way for the updated program to continue to provide old-version functionality. Our conformable specifications are inspired by simulation objects.

The connection between our updates that preserve backward compatibility and Ajmani's invariants can be captured in the following way. Let s_1 and s_2 be server programs, π a patch, and ϕ a backward-compatible client specification. For simplicity, suppose that s_1 and s_2 define the same set of functions and that update only ever appears as the last command in the body of a function. An *update invariant* is a binary relation I on server states such that

1. for all functions f, the evaluations of f in s_1 and s_2 from states related by I with an identical argument either both diverge or both terminate and yield identical results and states again related by I.
2. the update establishes the invariant—i.e., for all states σ_1 and σ_2 such that σ_2 can be obtained by running the state transformer starting from σ_1, we have that σ_1 and σ_2 are related by I.

It is not hard to show that if such an update invariant exists for π, then π preserves ϕ.

Liskov and Wing's work on behavioral subtyping [21] captures a condition like our notion of preserving a backward-compatible specification. They present a notion of subtyping that uses pre-conditions, post-conditions, and invariants to capture richer notions of the behavior of the object than can be described using types alone. However, behavioral subtyping does not address updates that make semantic changes to interfaces (e.g., removing a method) or behaviors (e.g., fixing a bug).

We generalize recent work by Hayden et al. [14] on systematically *testing* dynamic software updates. Given tests that should pass for both the old and new versions, Hayden et al. tested every feasible test run that started the test at the old version and updated at some point during the run. They proposed a compression strat-

egy which rules out runs that are provably equivalent to ones that have already been tested. Systematic tests are akin to our backward-compatible specifications. Our paper focuses on specification and verification of properties that go beyond simple tests.

Our merging tool was inspired by KISS [28], a program transformation also meant to precede off-the-shelf analysis tools. The goal of KISS is to transform a multi-threaded program into a single threaded program in which all possible choices of when to perform a single context switch are made manifest in the program text. Thus KISS makes manifest a non-deterministic choice. Our merger follows the same idea, where the non-deterministic choice is when to perform the update. The merging strategy for our application encodes the behavior of the original program exactly (cf. Theorem 1), whereas the KISS transformation only represents some of the possible executions of the original (those with one context switch).

8. Summary and future work

In this paper we have presented the first system for specifying and verifying the correctness of dynamically updated programs. We proposed client-oriented *update specifications* as a way to prescribe the correct behavior of updating executions, and formalized the syntax and semantics for these specifications. We showed how to construct update specifications for several common classes of updates based on existing, single-version specifications for the old and/or new programs. We developed a verification methodology based on combining the old and new programs into a single merged program, which allows us to check properties of updated programs using standard program analysis tools. We wrote a merging tool for C programs, and used it in conjunction with three analysis tools —Otter, a symbolic executor; THOR, a heap analysis tool, and Interproc, an abstract interpreter— to verify update specifications for several interesting C programs.

Our work on specifying and verifying properties of dynamic updates is ongoing. In the immediate future, we plan to expand the empirical study of our technique on more programs, to further evaluate its applicability, and more carefully understand the sources of overhead. For the longer term, we would like to develop technology that would permit us to check specifications directly, in terms of client/server interactions via sockets. One possibility would be develop a symbolic executor that can execute client and server programs together and accurately model interactions via I/O.

References

[1] The ASTREE static analyzer. `http://www.astree.ens.fr/`.

[2] Cassandra API overview. `http://wiki.apache.org/cassandra/API`.

[3] Redis - project hosting on Google Code. `http://code.google.com/p/redis/`.

[4] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *ECOOP*, July 2006.

[5] Jeff Arnold and Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Eurosys*, 2009.

[6] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, Laboratory for Computer Science, The Massachussets Institute of Technology, March 1983.

[7] Toby Bloom and Mark Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2): 102–108, March 1993.

[8] Gilad Bracha. Objects as software services. `http://bracha.org/-objectsAsSoftwareServices.pdf`, August 2006.

[9] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *ICSE*, 2007.

[10] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

[11] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[12] Deepak Gupta and Pankaj Jalote. On line software version change using state transfer between processes. *SPE*, 23(9), 1993.

[13] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE TSE*, 22(2), 1996.

[14] Christopher M. Hayden, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Efficient Systematic Testing for Dynamically Updatable Software. In *HOTSWUP*, 2009.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM Press, 2002.

[16] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6), 2005.

[17] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003. ISBN 0-321-22862-6.

[18] Cristian Kadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassistaed and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[19] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE TSE*, 16(11), 1990.

[20] Gaël Lalire, Mathias Argoud, and Bertrand Jeannet. Interproc, 2010. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`.

[21] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16:1811–1841, November 1994.

[22] 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.

[23] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, 2010.

[24] Kristis Makris and Rida Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX ATC*, 2009.

[25] Iulian Neamtiu and Michael Hicks. Safe and timely dynamic updates for multi-threaded programs. In *PLDI*, June 2009.

[26] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.

[27] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *LNCS*, 2304:213–228, 2002.

[28] Shaz Qadeer and Dinghao Wu. KISS: Leep it simple and sequential. In *PLDI*, 2004.

[29] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.

[30] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

[31] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates for Java: A VM-centric approach. In *PLDI*, 2009.

[32] Unsanity. Application Enhancer – enhance the applications by loading modules. `http://www.unsanity.com/haxies/ape`, 2010.

A. Proof of equivalence

For the proofs of the lemmas used, we change the transformation slightly, to make sure we always transform non-values into non-values. In particular, whenever $[\![e]\!]^{\cdot} = v$ where e is a non-value, we change the transformation so that $[\![e]\!]^{\cdot} = \mathsf{let}\ z = 0\ \mathsf{in}\ v$, and likewise for $\{\!|\cdot|\!\}^{\cdot}$ and $(\!|\cdot|\!)^{\cdot}$.

$$
\begin{array}{rcl}
(\!|\sigma|\!)^{p,\pi} & = & \{l \mapsto (\!|v|\!)^{p,\pi} \mid \sigma(l) = v\} \\
\hline
(\!|\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2|\!)^{p,\pi} & = & \mathsf{let}\ x = (\!|e_1|\!)^{p,\pi}\ \mathsf{in}\ (\!|e_2|\!)^{p,\pi} \\
(\!|\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2|\!)^{p,\pi} & = & \mathsf{if}\ (\!|v|\!)^{p,\pi}\ \mathsf{then}\ (\!|e_1|\!)^{p,\pi}\ \mathsf{else}\ (\!|e_2|\!)^{p,\pi} \\
(\!|v(v')|\!)^{p,\pi} & = & (\!|v|\!)^{p,\pi}((\!|v'|\!)^{p,\pi}) \\
(\!|\mathsf{update}|\!)^{p,\pi} & = & \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} \\
 & & \quad \mathsf{if}\ z\ \mathsf{then}\ 0\ \mathsf{else} \\
 & & \quad\quad \mathit{uflag} := ?; \\
 & & \quad\quad \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in} \\
 & & \quad\quad \mathsf{if}\ z\ \mathsf{then}\ (\{\!|e|\!\}^{p'}; 1)\ \mathsf{else}\ 0 \\
(\!|\mathsf{update}|\!)^{p,\pi} & = & 0 \\
(\!|\mathsf{running}\ p|\!)^{p,(p',e)} & = & 0\ \text{ or }\ \mathsf{let}\ z = \mathit{isupd}()\ \mathsf{in}\ z = 0 \\
(\!|\mathsf{running}\ p'|\!)^{p,(p',e)} & = & 1\ \text{ or }\ \mathit{isupd}() \\
(\!|\mathsf{running}\ p''|\!)^{p,(p',e)} & = & 0 \qquad p'' \neq p \wedge p'' \neq p' \\
(\!|g|\!)^{p,(p',e)} & = & g_{ptr} \quad \text{if } p(g) = \lambda x.e \\
(\!|g|\!)^{p,(p',e)} & = & g' \quad\ \ \text{if } p'(g) = \lambda x.e \\
(\!|l|\!)^{p,\pi} & = & l \\
(\!|v_1 := v_2|\!)^{p,\pi}, \text{etc.} & = & \text{transform } v_i \text{ recursively, as for if}
\end{array}
$$

Figure 12. Old/new program transformation.

$$
\begin{array}{rcl}
(p, \sigma_s) \rhd \pi & \triangleq & (\overline{p}, \overline{\sigma_s}) \\
\langle p; \sigma_s; e\rangle \rhd \pi & \triangleq & \langle \overline{p}, \overline{\sigma_s}, \overline{e}\rangle \\
\langle p; \sigma_s; \sigma_c; e\rangle \rhd \pi & \triangleq & \langle \overline{p}, \overline{\sigma_s}, \overline{\sigma_c}, \overline{e}\rangle \\
\langle p; \sigma_s; e\rangle\ [\rhd]\ \pi & \triangleq & \langle \overline{p}, \hat{\sigma_s}, \hat{e}\rangle \\
\langle p; \sigma_s; \sigma_c; e\rangle\ [\rhd]\ \pi & \triangleq & \langle \overline{p}, \hat{\sigma_s}, \hat{\sigma_c}; \hat{e}\rangle
\end{array}
$$

where

$$
\begin{array}{rclcrcl}
\pi & = & (p', e') & & & & \\
i & \leq & 0 & & j & > & 0 \\
\overline{p} & = & \{\!|p'|\!\}^{p'}, [\![p]\!]^{p,\pi} & & \hat{\sigma_s} & = & (\!|\sigma_s|\!)^{p,\pi}[\mathit{uflag} \mapsto j] \\
\overline{\sigma_s} & = & [\![\sigma_s]\!]^{p,\pi}[\mathit{uflag} \mapsto i] & & \hat{\sigma_c} & = & (\!|\sigma_c|\!)^{p,\pi} \\
\overline{\sigma_c} & = & [\![\sigma_c]\!]^{p,\pi} & & \hat{e} & = & (\!|e|\!)^{p,\pi} \\
\overline{e} & = & [\![e]\!]^{p,\pi} & & & &
\end{array}
$$

Figure 11. Transforming configurations.

A.1 Soundness

The first lemma states that the merged program simulates the original. To make the lemmas easier to read, we sometimes write $\langle p; C\rangle$ to represent an arbitrary configuration $\langle p; \sigma_s; \sigma_c; e\rangle$, where the latter three elements are not interesting. We write C is not an error state to mean that the e portion of C is not error.

Lemma 1. *For all p, p'', C, C'', π, where $\pi = (p', e)$, we have $\langle p; C\rangle \xrightarrow{\vec{\nu}}{}^{*} \langle p''; C''\rangle$ implies that*

- *$\vec{\nu} = \epsilon$ implies $p'' = p$ and $\langle p; C\rangle \rhd \pi \longrightarrow^{*} \langle p; C''\rangle \rhd \pi$*
- *$\vec{\nu} = \pi$ implies $p'' = p'$ and $\langle p; C\rangle \rhd \pi \longrightarrow^{*} \langle p; C''\rangle\ [\rhd]\ \pi$*

In words, this lemma states that any trace, which may have an update or not, taken by an untransformed program is matched by a trace in the transformed program. The lemma refers to $\langle p; C\rangle\ [\rhd]\ \pi$, which is the "post-update" transformation, shown in Figure 11. The definition is the same as $\cdot \rhd \cdot$ but for two differences. First, *uflag* is set to some positive integer, to indicate the update has occurred. Second, rather than transform the active expression and heap using $[\![\cdot]\!]^{p,\pi}$ it uses $(\!|\cdot|\!)^{p,\pi}$, where $(\!|\cdot|\!)^{\cdot,\cdot}$ is the union of the transformations $[\![\cdot]\!]^{\cdot}$ and $\{\!|\cdot|\!\}^{\cdot}$. That is, $(\!|e|\!)^{p,p'}$ applies either $[\![\cdot]\!]^{p,\sigma}$ or $\{\!|\cdot|\!\}^{p'}$ to e's outermost form, and then independently applies one or the other transformation to each of its subexpressions. The full transformation is shown in Figure 12. Using $(\!|\cdot|\!)^{\cdot,\cdot}$ is necessary because after the simulated update takes place locations l may end up binding either old/new function pointers f_{ptr} or new function pointers f'. Notice that if $[\![e_0]\!]^{p,\pi} = e'$ then $(\!|e_0|\!)^{p,\pi} = e'$ and likewise if $\{\!|e_0|\!\}^{p} = e''$ then $(\!|e_0|\!)^{p,\pi} = e''$; the same holds for transformed heaps.

This lemma is proved in three steps using the following lemmas, the first of which proves the simulation holds prior to an update, when taking a single step.

Lemma 2. *For all p, C, C', π, it is the case that $\langle p; C\rangle \longrightarrow \langle p; C'\rangle$ implies $\langle p; C\rangle \rhd \pi \longrightarrow^{+} \langle p; C'\rangle \rhd \pi$.*

The next lemma proves the simulation is also preserved by an update from the old program to the new one.

Lemma 3. *For all p, C, C', π, where $\pi = (p', e'')$, it is the case that $\langle p; C\rangle \xrightarrow{\pi} \langle p'; C'\rangle$ implies $\langle p; C\rangle \rhd \pi \longrightarrow^{+} \langle p; C'\rangle\ [\rhd]\ \pi$.*

The next lemma proves that the simulation is preserved following the update.

Lemma 4. *For all p, C, C', π, where $\pi = (p', e)$, it is the case that $\langle p'; C\rangle \longrightarrow \langle p'; C'\rangle$ implies $\langle p; C\rangle\ [\rhd]\ \pi \longrightarrow^{+} \langle p; C'\rangle\ [\rhd]\ \pi$.*

The proofs of the three previous lemmas involve proving isomorphic lemmas that apply to just the server semantics.

A.2 Completeness

We state completeness as follows:

Lemma 5. *For all p, p'', C, C'', π where $\pi = (p', e)$ it is the case that $\langle p; C\rangle \rhd \pi \longrightarrow^{*} \langle p''; C''\rangle$ implies there exists C' such that*

- *$\langle p''; C''\rangle \longrightarrow^{*} \langle p; C'\rangle \rhd \pi$ and $\langle p; C\rangle \longrightarrow^{*} \langle p; C'\rangle$; or*
- *$\langle p''; C''\rangle \longrightarrow^{*} \langle p; C'\rangle\ [\rhd]\ \pi$ and $\langle p; C\rangle \xrightarrow{\pi}{}^{*} \langle p'; C'\rangle$; or*

This lemma states that for any execution trace of the transformed program, there will be a corresponding trace of the untransformed program. However, the transformed program may need to execute a little more before it matches up with an untransformed state. The proof follows by repeated applications of the next two lemmas.

Lemma 6. *For all p, p'', C, C'', π where $\pi = (p', e)$ suppose that $\langle p; C\rangle \rhd \pi \longrightarrow^{+} \langle p''; C''\rangle$ and there exists no C_0 such that either*

- *$\langle p; C\rangle \rhd \pi \longrightarrow^{+} \langle p; C_0\rangle \rhd \pi \longrightarrow^{+} \langle p''; C''\rangle$ and $\langle p; C\rangle \longrightarrow \langle p; C_0\rangle$ or*
- *$\langle p; C\rangle \rhd \pi \longrightarrow^{+} \langle p; C_0\rangle\ [\rhd]\ \pi \longrightarrow^{+} \langle p''; C''\rangle$ and $\langle p; C\rangle \xrightarrow{\pi} \langle p'; C_0\rangle$.*

Then there exists C' such that either

- *$\langle p''; C''\rangle = \langle p; C'\rangle \rhd \pi$ and $\langle p; C\rangle \longrightarrow \langle p; C'\rangle$; or*
- *$\langle p''; C''\rangle = \langle p; C'\rangle\ [\rhd]\ \pi$ and $\langle p; C\rangle \xrightarrow{\pi} \langle p'; C'\rangle$; or*
- *$\langle p''; C''\rangle \xrightarrow{\nu} \langle p''; C'\rangle$.*

In words, this lemma states that if we have a transformed program state that can take some number of steps, then either that state matches an untransformed state we can also step to, or we can take another step. If we are in the last case, we can repeatedly apply the lemma to eventually reach a matching state.

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

Lemma 7. *For all p, p'', C, C'', π where $\pi = (p', e)$ suppose that $\langle p; C\rangle\ [\rhd]\ \pi \longrightarrow^{+} \langle p''; C''\rangle$ and there exists no C_0 such that $\langle p; C\rangle\ [\rhd]\ \pi \longrightarrow^{+} \langle p; C_0\rangle\ [\rhd]\ \pi \longrightarrow^{+} \langle p''; C''\rangle$ and $\langle p'; C\rangle \longrightarrow \langle p'; C_0\rangle$. Then there exists C' such that either*

- $\langle p''; C''\rangle = \langle p; C'\rangle\ [\triangleright]\ \pi$ *and* $\langle p'; C\rangle \longrightarrow \langle p'; C'\rangle$ *; or*
- $\langle p''; C''\rangle \longrightarrow \langle p''; C'\rangle$.

The proofs of all of these lemmas involve proving isomorphic lemmas that apply to just the server semantics.

A.3 Equivalence

One last lemma to prove:

Lemma 8. *For all p, π, e, we have that $e \neq$ error if and only if $[\![e]\!]^{p,\pi} \neq$ error and $(\!|e|\!)^{p,\pi} \neq$ error.*

This simply states that error states are preserved by the transformation, and is proved easily by inspection.

Now, recall Theorem 1:

> For all p, σ, π, ϕ, it is the case that $(p, \sigma), \pi \models \phi$ *if and only if* $((p, \sigma) \triangleright \pi) \models [\![\phi]\!]^{p,\pi}$.

We can expand the definitions of satisfiability and get

> For all p, σ, p', e'', ϕ where $\pi = (p', e'')$, it is the case that for all $e, p'', \sigma_s, \sigma_c, \vec{\nu}$ where $\vec{\nu} = \epsilon$ or $\vec{\nu} = \pi$, we have $\langle p; \sigma; \cdot; \phi\rangle \xrightarrow{\vec{\nu}}{}^* \langle p''; \sigma_s; \sigma_c; e\rangle$ implies e is not error *if and only if* for all $e', p''', \sigma'_s, \sigma'_c$ we have $\langle p; \sigma; \cdot; \phi\rangle \triangleright \pi \longrightarrow^* \langle p'''; \sigma'_s; \sigma'_c; e'\rangle$ implies e' is not error.

Here is the proof. Consider the forward direction. Pick any run $\langle p; \sigma; \cdot; \phi\rangle \triangleright \pi \longrightarrow^* \langle p'''; \sigma'_s; \sigma'_c; e'\rangle$ of the transformed program. By Lemma 5 there exists some C' such that

- $\langle p'''; \sigma'_s; \sigma'_c; e'\rangle \longrightarrow^* \langle p; C'\rangle \triangleright \pi$ and $\langle p; \sigma; \cdot; \phi\rangle \longrightarrow^* \langle p; C'\rangle$. By assumption, the latter fact gives us that $\langle p; C'\rangle$ is not an error state and thus $\langle p'''; \sigma'_s; \sigma'_c; e'\rangle$ is not an error state either. This is because either $\langle p'''; \sigma'_s; \sigma'_c; e'\rangle = \langle p; C'\rangle \triangleright \pi$ and thus the result follows by Lemma 8, or because $\langle p'''; \sigma'_s; \sigma'_c; e'\rangle$ can take a step, which is not possible in an error state.
- $\langle p'''; \sigma'_s; \sigma'_c; e'\rangle \longrightarrow^* \langle p; C'\rangle\ [\triangleright]\ \pi$ and $\langle p; \sigma; \cdot; \phi\rangle \xrightarrow{\pi}{}^* \langle p'; C'\rangle$. Same argument as above.

Now consider the backward direction. Pick some evaluation $\langle p; \sigma; \cdot; \phi\rangle \xrightarrow{\vec{\nu}}{}^* \langle p''; \sigma_s; \sigma_c; e\rangle$. By Lemma 1

- $\vec{\nu} = \epsilon$ implies $p'' = p$ and $\langle p; \sigma; \cdot; \phi\rangle \triangleright \pi \longrightarrow^* \langle p; \sigma_s; \sigma_c; e\rangle \triangleright \pi$. By assumption, the latter fact gives us $[\![e]\!]^{p,\pi}$ is not error, and Lemma 8 e cannot be error either.
- $\vec{\nu} = \pi$ implies $p'' = p'$ and $\langle p; \sigma; \cdot; \phi\rangle \triangleright \pi \longrightarrow^* \langle p; \sigma_s; \sigma_c; e\rangle\ [\triangleright]\ \pi$. By assumption, $(\!|e|\!)^{p,\pi}$ is not error, and thus by Lemma 8 e cannot be error either.

QED.