XCVB: Improving Modularity for Common Lisp

François-René Rideau
ITA Software, Inc.
fare@tunes.org

Spencer Brody
Brown University
sbrody88@gmail.com

Abstract

We present XCVB, a new open source system to build Common Lisp software. XCVB notably features separate compilation. This departs from the Common Lisp tradition of compiling within a live world. But separate compilation scales to very large systems, and opens many possibilities for the future.

1. Introduction

Despite plenty of advanced module systems having been implemented in various Lisp dialects (and languages from other families) in the last 25 years, the current state of the art for the Common Lisp community hasn't improved much since 1980: a central file defines the build graph, and components are compiled and loaded as successive side effects to the current Lisp world.

Our insight is to break the classic Lisp assumption of programming in a single concrete Lisp world that is side-effected as source code is sequentially compiled then loaded. Instead, we propose a pure-functional approach of building encapsulated components, each compiled in its own isolated virtual Lisp world.

The traditional Lisp model may have been ahead of its time in the 1970s and valid into the 1980s, when memory and virtualization were expensive. But it doesn't scale well to programming in the large in the 2000s, with projects of tens of programmers with hundreds of files, where the side-effects of processing various files may interact in unexpected ways. Our model, which is traditional for many other languages (like OCaml), does scale to large projects and allows a concurrent and distributed build.

2. Past Common Lisp build systems

In the original DEFSYSTEM (Weinreb and Moon 1981) and its descendents, a "system" is defined in a central file as an acyclic graph of components to be compiled and loaded into the current Lisp world, arcs being dependencies.

The original DEFSYSTEM and its once-popular portable descendent MK:DEFSYSTEM (from 1990 on) required one to manually list all transitive dependencies of a component in a statement separate from its definition (see figure 1). The current state of the art and most popular descendent of DEFSYSTEM, ASDF (Barlow and contributors 2004), simplifies things tremendously by automatically computing the transitive closure of directly declared dependencies from single statements defining both both component and direct dependencies (see figure 2). Such improvements and more had been suggested as early as 1984 (Pitman 1984; Robbins 1985) but not available as a portable package until ASDF was created in 2001. ASDF also features some limited form of extensibility using CLOS; but for the purpose of this presentation, we will focus on the core architecture of the build system. Another DEFSYSTEM variant, mudballs, was recently created, that seems to cleanup and simplify ASDF while keeping the same general architecture.

```
(:module BAR ("BAR"))
...
(:compile-load BAR
   (PACKAGES MACROS SPECIALS)
   (:fasload PACKAGES MACROS SPECIALS))
```

Figure 1. Using DEFSYSTEM: component `bar` is defined in two places in system file `foo.system` – all transitive dependencies have to be listed explicitly.

```
(:file "bar" :depends-on ("macros" "specials"))
```

Figure 2. Using ASDF: component `bar` is defined somewhere in system file `foo.asd` – transitive dependency on `packages` may be skipped.

```
#+xcvb
(module
 (:depends-on ("macros" "specials")))
```

Figure 3. Using XCVB: component `bar` is defined on top of module file `bar.lisp` – transitive dependency on `packages` is implicit.

3. XCVB

XCVB, the eXtensible Component Verifier and Builder, is a new open source system to build software written in Common Lisp.

XCVB most notably features separate compilation in isolated Lisp worlds. The current implementation spawns a new Lisp process for each compilation. In XCVB, dependency information is not declared in a central file, but at the top of each component file (see figure 3).

In the following bullet-point presentation, we briefly outline the reasons why separate compilation is important, though it requires a break from the traditional Common Lisp model. We describe the state of the project and the benefits at hand. We suggest the many future benefits that can be enabled by this design, and we discuss the challenges that will have to be overcome to achieve some of those benefits.

4. XCVB yesterday

What the XCVB prototype already brings as compared to ASDF.

4.1 Goal: Separate Compilation

- Independent compilation of individual files.
- Compute objects from source, just as in any modern language.
- Proper staging of compile-time dependencies (macros compiled before they are used).
- Semantics of a file fully encapsulated in its contents (+ contents of dependencies).
- Incremental change-driven building and testing.

4.2 Therefore: Dependencies must be declared locally

- Move dependencies away from centralized off-file meta-data.
- Module import statement, just as in any modern language.
- No more global recompilation (or subtle failure) at the least change in centralized system definition.
- No more subtle bugs due to non-local change in ordering of compile-time side-effects (see figure 4).
- Unlike ASDF, can incrementally track dependencies across systems.

4.3 Eager enforcement of dependencies

- Build each file in a world loaded with none but its dependencies.
- Requires import discipline, just as in any modern language.
- A bit slower, but much more robust: dependency bugs are detected early (as opposed to figure 4).
- No more unmaintainable large manual dependency graphs (or rigid unmaintained serial lists of files as a workaround).
- Allows correct incremental unit tests based on what has changed (beware: tests that use reflection).

4.4 Current build backends

- XCVB computes the build graph, currently lets other software do the build.
- `Makefile`: integrate into a larger build, just as in any modern language.
- ASDF: integrate into legacy ASDF builds.
- More backends possible in the future: `OMake` (Hickey and Nogin 2006); take over your build.

4.5 Decoupling builder and buildee

- Protection from uncontrolled side-effects from buildee to builder.
- Allows for integration with `make` as mentioned above.
- Allows for cross-compilation from one compiler/architecture to a different one.
- Allows for a feature-rich build system that needn't fit in one small file, yet.
- Allows builder to rebuild and test its dependencies and self.

4.6 Can use CFASLs to capture COMPILE-TIME side-effects

- Vast speed improvement, fewer rebuilds (the FASL may have changed yet the CFASL stay the same).
- Like C++ precompiled headers, except automatically deduced from the code.

Step 1: Initially working system.

```
;; bar provides base for baz
(:file "bar" :depends-on ("base"))

;; quux unrelatedly depends on base
(:file "quux" :depends-on ("base"))

;; baz also depends on base,
;; transitively provided by bar
(:file "baz" :depends-on ("bar"))
```

Step 2: after refactoring, `bar` is simplified to not need `base`

```
;; bar no longer depends on base
(:file "bar" :depends-on ("packages"))

;; quux happens to load base
;; before baz is compiled
(:file "quux" :depends-on ("base"))

;; baz really depends on base
;; but the missing dependency is not detected
(:file "baz" :depends-on ("bar"))
```

Step 3: `quux` is also refactored to not need `base`. Now compilation of unrelated file `baz` breaks because its *implicit* dependency `base` is not loaded anymore.

```
;; bar still does not depend on base
(:file "bar" :depends-on ("packages"))

;; quux no longer loads base
(:file "quux" :depends-on ("packages"))

;; despite a lack of related modifications,
;; baz now breaks inexplicably
;; at next clean compilation
(:file "baz" :depends-on ("bar"))
```

The lines that matter need not be consecutive but may be separated arbitrarily. The missing dependency may not be a direct dependency, but any transitive dependency. Steps 2 and 3 may be separated by a lot of unrelated changes. Also, they need not be done by the same person, rewarding the culprit, punishing the innocent, encouraging sloppiness and discouraging refactoring.

Figure 4. Break down of ASDF: a mistake at step 2 creates a timebomb that is only triggered by an innocent change much later.

- Was easily added to SBCL by Juho Snellman, could be as easily added to other compilers.
- Careful `eval-when` discipline required (as with ASDF really, but now it is enforced).

4.7 Automated migration path from ASDF

- XCVB accepts dependencies from XCVB systems to ASDF systems and vice-versa.
- Automatic migration of your ASDF system using Andreas Fuchs's `asdf-dependency-grovel`.
- Compile-time Lisp state requires extending the dependency-detection tool.
- ASDF extensions will require according XCVB extensions.

5. XCVB today

Urgently needed.

5.1 User friendliness

- Add documentation and examples.
- Better behavior in face of errors.

5.2 More features

- Combine multiple projects, find them using a search path.
- Refine migration and compilation to deal with harder cases (data files read at compile-time, computed lisp files, etc.).
- Have a more general model for staged builds (multiple intermediate images, dynamic dependency computation).

5.3 Actually migrate a critical mass of existing ASDF systems

- Support manual overrides when automation breaks down.
- Maintain until upstream adopts XCVB (if ever) – automated migration makes that possible.
- Provide a distribution system (as in `asdf-install`, `mudballs` or `clbuild`, etc.).
- Fully bootstrap XCVB (make ASDF optional).

5.4 Refactor Internals

- Current implementation was a good first attempt, but needs to be reworked.
- Needs to be made more general to allow for desired and future features.
- Recognize hand-coded patterns, read literature, formalize a domain, grow a language.

6. XCVB tomorrow

The following improvements are enabled by XCVB's deterministic separate compilation model.

6.1 Distributed backends

- Pluggable distributed compilation (`distcc` for CL).
- Take over the build, make it distributed with Erlang-in-Lisp.
- Requires compiler support to preserve source locations for debugging.

6.2 Caching

- Cache objects rather than rebuild (`ccache` for CL).
- Base cache on crypto hash fully capturing the computation and its dependencies.
- Can track all the modified dependencies since last success at building and verifying a component.
- Push for more determinism in Lisp compilers!

6.3 Dependency management

- `xcvb-dependency-check` to detect superfluous dependencies (to be based on `asdf-dependency-grovel`).
- Cache above results to suggest missing dependencies.
- Actually implement dependency-based testing.
- Integrate test dependency tracking with code-coverage tools.

6.4 Extend the build Specification Language

- Build rules that call arbitrary programs (as in a `Makefile`).
- Computed source files, including from parametrized computations.
- Dependency on arbitrary computed features, only compiled once.
- Automated finalization and verification of modules.

6.5 Manage reader extensions, alternate grammars, hygienic macros, etc.

- Made possible and convenient by separate compilation.
- No pollution of compile-time environment from other modules.
- Everyone can use whatever fits his purposes, with well-defined semantics.
- Requires compiler support to preserve source locations for debugging.

6.6 Layer namespace management on top of it

- Automate evolution of `defpackage` forms.
- More sensible replacement for packages (lexicons? modules as in PLT Scheme?).
- Higher-order parametric components (PLT Scheme units).
- Many levels of static typing with interface that enforces implicit contracts, etc.
- Generally, make CL competitive again wrt access to latest improvements from research.

6.7 Abstract away the execution model

- Semantics: proper tail calls? continuations? serializable state? etc.
- Performance: debuggability? optimization levels?
- A file can require some of the above settings.
- A same module can be compiled according to many combinations of them.

7. Need for extensions to the CL standard

Short of reimplementing all of CL in a translation layer, some of the above features cannot be implemented on top of standard CL: they require access to functionality below the standardized abstraction barrier of a CL implementation.

7.1 Access to system functions

- open, fork, exec, sockets, etc. – happily we have CFFI, IOLib.
- nothing specific to XCVB here, but still (sadly) deserves mentioning.

7.2 Encapsulation of COMPILE-TIME side-effects

- CFASL only in SBCL for now.
- slow loading "FAS"L can make do if you can cope with intermixing LOAD-TIME side-effects.

7.3 Encapsulation of LOAD-TIME partial state, not side-effects

- FASL is still too slow to load, cannot be shared between binaries.
- `SB-HEAPDUMP` can be `mmap`()ed – but isn't even standard feature of SBCL.

7.4 Programmable access to debugging meta-information

- Syntax extension requires support for recording source locations.

- Semantic layering is a challenge for single-stepping, access to high-level view of the state.
- Support multiple evaluation models in a given running environment.

7.5 PCLSRing

Generalizing PCLSRing (Bawden 1989), when interrupting, inspecting or single-stepping a program in a higher-level language, you don't want to handle intermediate states of the low-level implementation, but safe points with a meaningful high-level interpretation.

- Needed for transactionality in single-stepped and/or concurrent evaluation.
- Challenge: a good meta-level protocol for users to define PCLSRing for arbitrary semantic layers.
- With such a tool, all the system can be implemented with first-class translation layers.

8. Conclusion

XCVB is nothing fancy – just elaborate plumbing. The ideas within are mostly well-known; each but the most prospective of them are already implemented in many build or module systems for other languages. Yet the bulk of the work is still ahead for XCVB. That's how far behind Common Lisp is with respect to modularity.

The deep rationale for XCVB is a social concern: minimizing programmer-side cognitive burden in combining modules. Technical and social aspects are tied in obvious ways, yet most people wilfully ignore at least one of the two aspects.

XCVB was initially developed by Spencer Brody during the Summer 2008 at ITA Software, under the guidance of François-René Rideau. Rideau briefly worked on it in mid December 2008 to release and document a usable prototype.

Our hope is that by the time the conference happens, we will have already deployed XCVB on a large system, and moved some points from "XCVB today" into "XCVB yesterday". However as of this writing, this hasn't happened yet.

XCVB code and documentation can be found at:

`http://common-lisp.net/projects/xcvb/`

Acknowledgments

This presentation describes work done at ITA Software, Inc.
Many thanks to James Knight for the essential insights.
Thanks to Juho Snellman for CFASL support in SBCL.

References

Daniel Barlow and contributors. *ASDF Manual*, 2004. URL `http://common-lisp.net/project/asdf/`.

Alan Bawden. PCLSRing: Keeping Process State Modular. Technical report, MIT, 1989. URL `http://fare.tunes.org/tmp/emergent/pclsr.htm`.

Jason Hickey and Aleksey Nogin. OMake: Designing a scalable build process. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006*, pages 63–78. Springer, 2006.

Kent Pitman. The Description of Large Systems. MIT AI Memo 801, September 1984. URL `http://www.nhplace.com/kent/Papers/Large-Systems.html`.

Richard Elliot Robbins. BUILD: A Tool for Maintaining Consistency in Modular Systems. MIT AI TR 874, November 1985. URL `ftp://publications.ai.mit.edu/ai-publications/pdf/AITR-874.pdf`.

Dan Weinreb and David Moon. *Lisp Machine Manual*, 1981.