YAM- A Framework for Rapid Software Development

Abhinandan Jain and Jeffrey Biesiadecki
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA 91109

***Abstract*— YAM is a software development framework with tools for facilitating the rapid development of software in a concurrent software development environment. YAM provides solutions for thorny development challenges associated with software reuse, managing multiple software configurations, developing of software product-lines, and multiple platform development and build management. YAM uses release-early, release-often development cycles to allow developers to incrementally integrate their changes into the system on a continual basis. YAM facilitates the creation and merging of branches to support the isolated development of immature software to avoid impacting the stability of the development effort. YAM uses modules and packages to organize and share software across multiple software products. It uses the concepts of link and work modules to reduce sandbox setup times even when the code-base is large. One side-benefit is the enforcement of a strong module-level encapsulation of a module's functionality and interface. This increases design transparency, system stability as well as software reuse. YAM is in use by several mid-size software development teams including several developing mission-critical software.**

I. Introduction

Managing change is a key requirement for successful software development. Change can be driven by new requirements, design evolution, refactoring needs, bug fixes etc. While change may be the one constant during development, it can also create instabilities in the development process and the software. An acceptable development pace is one where the software development activity remains stable, i.e. the team is productive and the quality of the software remains high. The quality of the software development infrastructure and processes providing structure for a development effort also determine the development pace. There is a fine line between having too little structure, leading to development instability, and too much structure, creating excessive overhead and constraints on the development effort.

As illustrated in Figure 1, the scope of software development efforts can range from a single developer developing a single software product to one involving distributed teams developing multi-product software. Key software development challenges include coordination of changes across the team, version control, handling large code-bases, integration and testing, multiple target development, software sharing etc. Yard sticks for measuring the success of a software development activity are the quality of the software itself, and the development pace.

Brooks' *The Mythical Man-Month* [1] was one of the early influential essays to analyze the software development process.

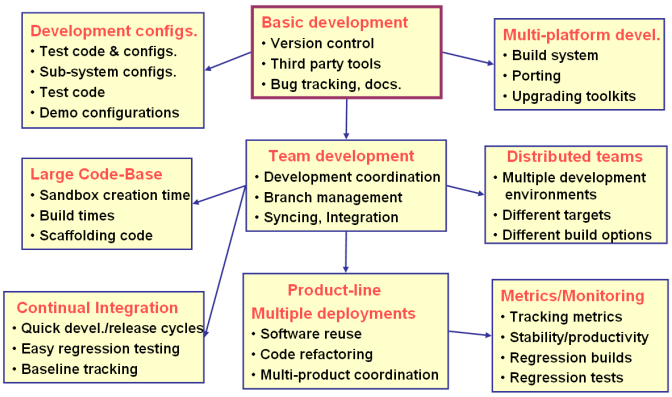

Fig. 1. Levels of software development environments ranging from individual developer/single product to distributed team with product-lines..

Brooks argues that the dynamics of software development is such that contrary to conventional wisdom, adding more developers to a team late in a project can delay rather than speed up the completion of the project. He highlights the need for getting the design right - asserting that the project should be willing to throw away the first implementation. Eric Raymond's *Cathedral and the Bazaar* and other essays [2] study the open-source software development model. Eric lists several insights to explain the success of the "bazaar" style open-source development in contrast with the "cathedral" style based on the principles of command and discipline. There have also been a series of software development processes referred to as *agile software development* [3] which advocate short iteration cycles. One of the best known examples of agile development is the *Extreme Programming* style [4] which requires that all development be done by pairs of developers, and that test cases be written before starting software implementation.

A. Motivation for YAM

The initial version of **YAM** [5] (*Yet Another Make*), developed a little less than a decade ago, was motivated by the need for processes to manage a multi-product simulation software development effort involving a small team at JPL's DARTS Lab [6]. The simulation end-products were quite diverse and driven by different mission testbed needs. The development team's strategy was to develop and share a pool of software models and algorithms. The team had the task of engineering the core toolkit so it remained adaptable and reusable across different

domains, while simultaneously meeting the custom functional and interface needs of end-users. With some of the deployment being in mission-critical testbeds, a significant level of rigor and discipline was necessary to produce high quality and maintainable simulation products. The diversity and evolving nature of the end-products required continual improvements in and refactoring of the core toolkit capabilities. It was clear that processes that went well beyond basic version control and good design were needed to support and sustain the development nimbleness and software sharing required by the team. Over time, other software development teams have adopted **YAM** to support their development needs and have driven the evolution and maturation of **YAM** itself.

YAM advocates continual development and integration cycles in the spirit of agile development. However, with **YAM**, software sharing is not a luxury but is rather regarded as fundamental to good software design and a sound development process itself. This perspective underlies several novel software concepts in **YAM**. While **YAM** is meant for an organized team-based software development effort, it does contain elements of bazaar style decentralized development. **YAM** integrates elements from areas such as configuration management, concurrent development, build management, software organization, and software reuse.

YAM provides solutions to challenges facing software development depicted in Figure 1 such as: continual software integration and test, concurrent development across distributed teams, development of software product-lines, multiple platform development, handling large code-base etc. For example, **YAM**'s use of "link modules" allows even large developmental sandboxes to be built up easily and quickly. This helps developers get used to routinely creating and abandoning sandboxes as a matter of course. Such agility is in sharp contrast to the not uncommon situation of developers hanging onto their functional sandboxes for long periods of time fearing long build times for new sandboxes, or of crippling the sandboxes through a merge! Also, by making the creation and merging of branches easy and transparent to users, **YAM** facilitates smooth and stable concurrent development. This paper describes the ideas and principles which form the basis of the **YAM** software development framework.

B. Software Development Needs

The **YAM** software development paradigm is guided by a "hierarchy of needs" (illustrated in Figure 2) underlying software development efforts. Colloquially, this hierarchy suggests that we cannot run before learning to walk, or walk before learning to crawl. This hierarchy is in the spirit of the hierarchy of needs developed by Maslow [7] in the arena of human psychology[1] to explain what drives human behavior. Maslow posited that human beings are incapable of addressing a higher level need until the lower level ones have been satisfied. This holds true for software development as well. Starting from the lowest level, the software development needs are:

[1]Maslow's needs hierarchy ranges from: physiological needs, to survival needs, to belongingness needs, to esteem needs, and finally to actualization needs.

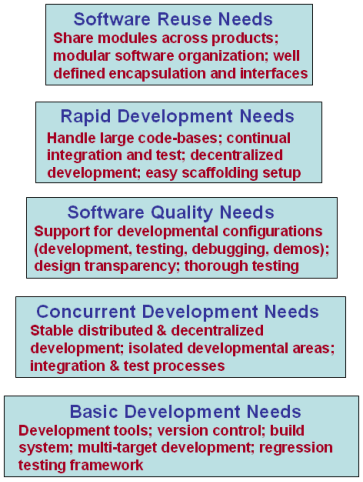

Fig. 2. Hierarchy of needs underlying software development

Basic Development Needs: This is the lowest level representing the minimal needs for a functional development environment. These needs encompass development tools (eg. compilers, debuggers, third party software etc.) to support software development; a software version-control and configuration management system; processes to rollback software; build systems; regression testing setup; support for multi-platform development; bug-tracking; documentation. These needs apply to even individual software development.

Stable Concurrent Development Needs: This class of needs is for development efforts involving more than one developer. Concurrent development requires processes to manage, coordinate and merge changes from parallel development within a team. During the course of development, developers need access to stable versions of the rest of the system while at the same time shielding the rest of the team from their immature developmental software. Processes for merging and syncing up changes need to be clearly defined. Additional coordination may be needed if the team is distributed across remote sites. Sound processes are needed to avoid instability and confusion during concurrent development.

High Quality Software Needs: Beyond stable concurrent development are the needs for developing high quality software - as measured by performance, robustness, maintainability and defect rates. Meeting these needs requires extensive integration and testing of the software at the unit and system levels. Keys to improving the software quality is in depth, multiple-levels of testing of the software and adjusting the design and implementation based on the test results and user feedback. Testing (as well as development) at the unit, sub-system and system level require the ability to create software configurations tailored for these purposes. The software development process needs to facilitate the easy creation and use of such configurations as a natural part of the development process. It is inevitable that such configurations will be created, and without built-in support, a significant part of the team effort can be wasted in creating brittle, custom environments to meet these needs.

Rapid Development Needs: Having met the needs for a high quality software development, the next level of needs is geared

towards increasing the team's productivity and development pace. Increasing the team development pace requires processes to: reduce integration and test times; reduce build and development times as the system code base gets large; support fast set up of scaffolding software; decentralize development and reduce coordination overhead; and facilitate easy prototyping and refinement of design concepts.

Reuse Needs: While the previous needs focused on a "single" software product, it is often the case that the team or the organization is responsible for multiple software products, i.e. a software product-line. For cost-effectiveness, such product-lines can involve significant software sharing across the different products. Effective software sharing that goes well beyond the designing software for reuse. When products A and B share software C, a development process that strait-jackets the evolution of the shared software C is destined to end up with variants of C within A and B. Such forking can leave a worse situation than if A and B had started off with independent implementations of C. Effective sharing requires a development process that can accommodate the evolution of software shared among multiple products. Changes to such shared software may be driven by internal refactoring needs, availability of new software technology, new platforms, new performance drivers or by requirements that trickle down from other products.

The above hierarchy of needs is useful for assessing the capabilities and gaps in a software development process. An implication of the needs hierarchy is that efforts to satisfy higher-level needs without having addressed the lower level needs are doomed to failure. For instance, attempting to carry out concurrent software development without adequate regression testing or version control infrastructure is unlikely to be very effective. Similarly, it is pointless to attempt to increase development pace without first having a process that produces acceptable, high quality software. Often team development environments struggle to get past the first and second levels. It is not uncommon to see the development pace being throttled back to a slow deliberate pace with extensive coordination overhead to help provide stability to the development process. As a consequence, software sharing is often little more than an after thought in such an environment.

While providing an analysis framework, the needs hierarchy does not however provide a prescription for designing a software development processes using these tiers, nor does it imply that a successful software development model is separable into distinct layers meeting the different tiers. A software development process needs to be viewed in its organic whole - with its internal dynamics determined by the development scope and the team, with the needs hierarchy providing the means for periodically assessing and improving the processes.

Our experience with the **YAM** development model has been that concepts that might be perceived - and even at times conceived - for higher-level needs, often end up significantly helping meet lower level needs as well. **YAM**'s organization of all software into modules (pg. 4), while necessary for sharing software across products, also turns out to be ideal for creating developmental software configurations that are essential during development. **YAM** modules also improve software modularity, and help better manage the mix of stable, and new and immature, portions of the code base. **YAM**'s use of link modules (pg. 5) for the rapid creation of user sandboxes (pg. 5) has the side-effect of enforcing strong module-level encapsulation which enforces software organization with significantly reduced coupling among the modules. In turn, this helps increase design transparency and software quality. **YAM**'s use of conventions and tools for the easy creation and merging of module-level branches simplifies version control as well as concurrent development. Supporting such hierarchy inversion, is Raymond's observation [2] that software reuse serves to improve software quality by making the "grass taller, the more it is grazed". The remainder of this paper describes in detail the key ideas and concepts underlying **YAM** and how they address the software development hierarchy of needs.

II. Key Concepts from the YAM Framework

YAM provides a set of user commands, in the style of CVS [8], to support day-to-day development. Additional **YAM** commands are also available for administration support. As illustrated in Figure 3, **YAM** components include a source repository with version control, a database for storing release data, command line scripts and make rules for building software. The current **YAM** toolkit uses CVS for version control,

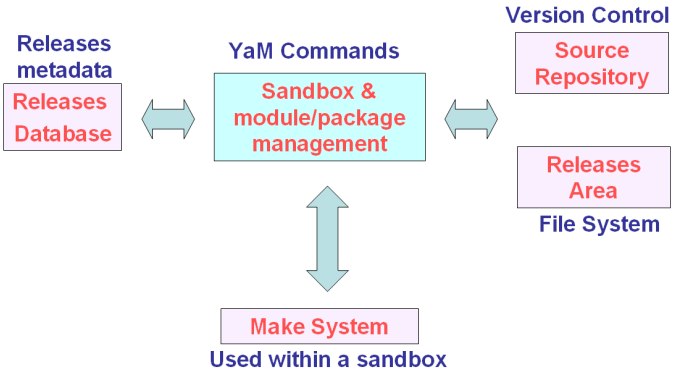

Fig. 3. The key components of **YAM** include a source repository, a database for releases data, and Perl for command line scripts.

MySQL for the database [9], PHP for Web interface [10] and Perl for the command line scripts [11]. **YAM** uses an abstract interface to these tools to facilitate its use with other tools such as Subversion.

This section describes some of the key design ideas and concepts that make up the **YAM** software development framework. The working model we use here is of a collocated team, or a group of collocated teams engaged in a software development activity. Here, collocation does not imply physical collocation but instead a coherent and shared file-system and development environment.

† **Low-level version control using CVS**

The **CVS** open source version control tool is used for low-level version control of the software. This includes the use of branches, tagging, handling of directory structures as well as remote client/server use.

† **All software is organized into YAM "modules"**

All software is broken up and organized into software units referred to as modules. A **YAM** module is simply a directory of

files, containing a file called **Makefile.yam** which implements a set of standard **YAM** make targets. The size, content and number of modules is completely up to the discretion of the development team.

† **Support software is also stored in modules**

Even support software, e.g. test harness, scripts, software stubs and utilities that support the development effort are managed within modules. Keeping such support software within version controlled modules gives the team easy access to them, and avoids the risk of "losing" them within developers' custom setups. For modules providing libraries, it is a common practice to include unit tests for the library within the module.

† **YAM "packages" are defined as module bundles**

As illustrated in Figure 4, **YAM** packages are simply bundles

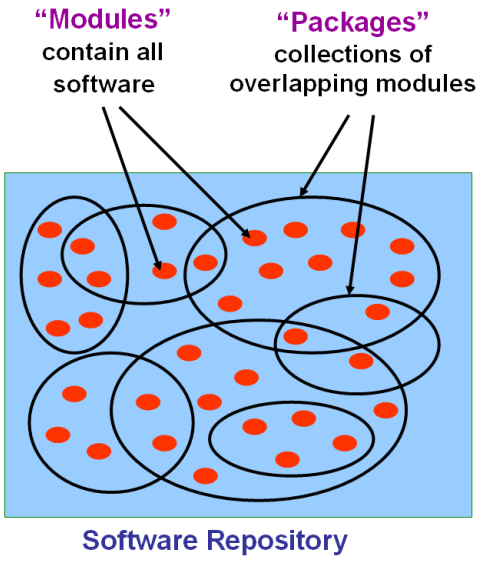

Fig. 4. All software being developed is organized into modules, with packages representing different bundles of modules.

of **YAM** modules and do not contain any additional software. Different packages can share modules between them, and can also contain other packages. While modules can provide component libraries, packages are typically runnable applications.

† **All module development occurs on branches**

Developers do all development on private branches. This isolates them from each others changes during development. The main trunk is used as the **release** branch. On conclusion of a development phase, developers release their changes by merging them onto the main trunk and committing and releasing the relevant modules. **YAM** provides commands to simplify the creation, merging and release of such branches and releases.

† **Development via fast branch/release cycles**

Frequent and regular releases of modules and packages are highly encouraged. Module development branches are expected to be short-lived and to be used to implement specific features or fixes. This is driven by two important goals. First, long life branches can require disproportionately large merge and integration effort on the part of the developer. Secondly, such large integration efforts can also require a large effort from the rest of the team to absorb and handle such large changes after they are released. **YAM**'s typical module branch/release cycles are illustrated in Figure 5.

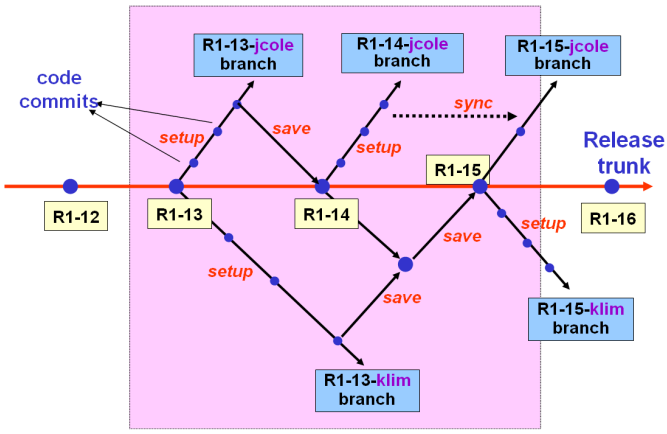

Fig. 5. The software development proceeds through a continual series of branch, develop, merge and integration cycles. The **YAM** commands in the figure are described in Sectino II-A.

† **Releases are managed at the module and package level**

Developers make releases of individual modules. On release, the source code for the module is tagged with a release number

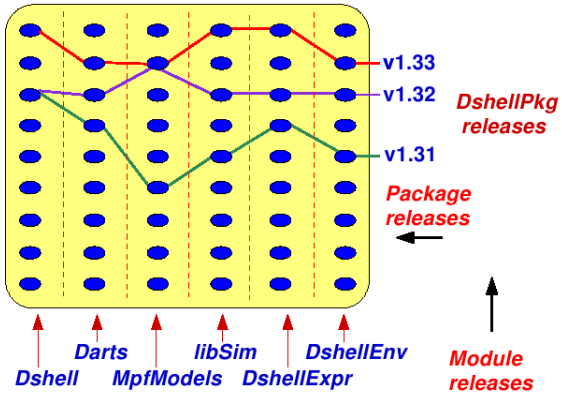

Fig. 6. The module release area hosts built versions of the releases of all the modules. Each column represents a module and the entries in the column the different releases for the module. package releases on the other hand are the lines that run horizontally across the modules and define specific module releases that belong to a package release.

for later use if needed. The fully built version of the module is stored for later use in a module release area. The typical requirement for a module release is that it pass the test suite. During a module release, **YAM** generates candidate entries for a module's ChangeLog file from log messages from individual file commits. Packages can also be released. Package releases simply consist of a specific grouping of module releases as illustrated by Figure 6. All release information is recorded in a database and a Web script provides on-line access to the information. While such short developmental cycles are the norm, there are times when specific developments require longer than usual development on isolated branches. Though not recommended, these are not precluded by **YAM**.

† **module releases are available as link modules**

On release, the built version of the module is moved from the

developer's sandbox into a module release area. They contain built versions of all the libraries and binaries provided by the module. The module release area is available to sub-teams sharing a development environment. These built versions can be used to quickly create virtual presence of these modules within developer sandboxes for use as scaffolding code and are hence referred to as as **link** modules.

† **All work is done in independent sandboxes**

All development is carried out within independent user "sandboxes". A user sandbox is a development directory

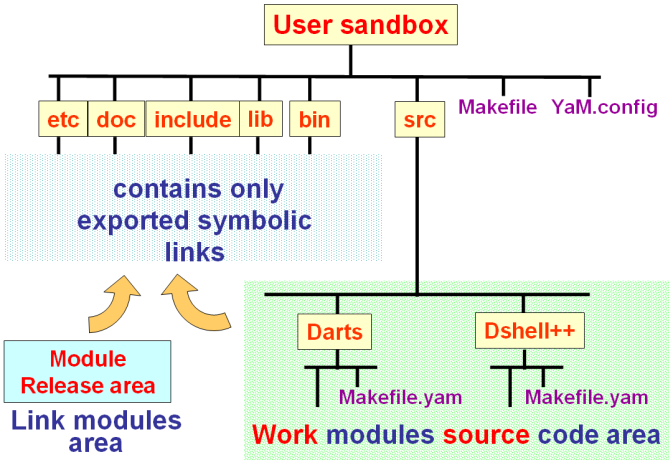

Fig. 7. The structure of a typical YAM sandbox.

created by the user containing the modules to be worked on with directory structure shown in Figure 7. The top-level of a sandbox contains a `src/` sub-directory for checkout out module source code. Other top-level directories such as `include`, `lib` etc. are referred to as the **export** area of a sandbox and only contain symbolic links from the modules. YAM provides commands to simplify the creation of private sandboxes for YAM packages. These commands also create branches for and checkout modules to be worked on within the sandbox. On completion of the development work, YAM commands can be used to make releases of the modules. Developers can create as many sandboxes as they need.

† **Developers can tailor a sandbox configurations**

Developers can tailor the modules contained in a sandbox. Moreover, they can specify which of the modules are present as simply **link modules** and which ones are **work modules**, i.e. ones whose source code is checked out into the `src/` directory of a sandbox. Each sandbox contains a `YAM.config` file that defines the the mix of link and work modules for the sandbox. As illustrated in Figure 8, a sandbox configuration can consist of only link modules (eg. for running regression tests) or all work modules (eg. for regression builds) or more typically, a hybrid mix. Usually, a developer will create a sandbox with work modules that he/she plans to work on, and the rest of the required modules as link modules to provide scaffolding code to complete the sandbox. Since link modules require just symbolic links they are much faster to create since they do not need to be checked out and built. The developer can also select the versions of link modules and module release branches to include in the sandbox. YAM also

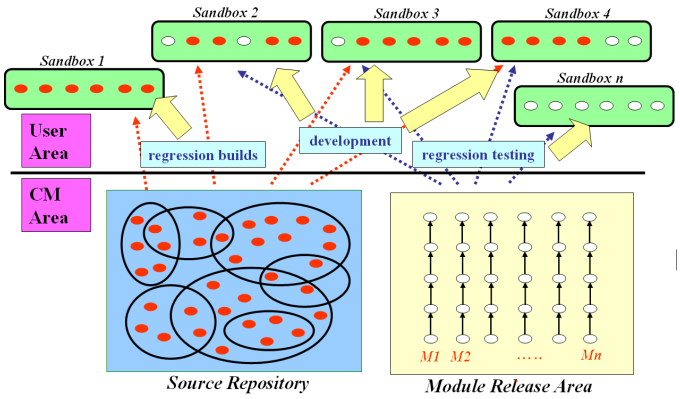

Fig. 8. Sandbox configurations can range from all work modules, to all link modules, to a mix of link and work modules.

provides support for updating a sandbox's configuration to add and remove work and link modules.

† **Explicit exporting of module interfaces**

To meet the requirement that modules be usable as work modules as well as link modules, all YAM modules export their public interfaces via symbolic links to a top-level **export** area within the sandbox. This allows module A to access files from module B irrespective of whether module B is a link or a work module in the sandbox. Thus, a module makes available its libraries to other modules by exporting links for them into the export area. On the other hand, if the module needs header files from other modules in order to build the library, then it looks for them in the export area and not directly in the module directory. The other modules should have exported links for these header files to the export area. The export area serves as a central nexus for interactions between the different modules. YAM provides make rules to simplify exporting of links by modules. There is support for physically copying such exported files for portability to non-Unix platforms that do not support symbolic links.

† **modules have a standardized build interface**

The only requirement on what constiutes a YAM module is that it provide a `Makefile.yam` make file that supports a small number of make rules for cleaning and building the module. Thus all modules have a uniform **make** file interface. YAM supports the generation of such makefiles from templates limiting the module specific customization largely to the definition of variables using supporting make files provided by a special **SiteDefs** module. modules can internally have arbitrary levels of additional build infrastructure as long as this minimal interface makefile requirement is satisfied. The module make interface establishes uniform conventions much like **autoconf** does for open source software.

† **Tracking third party tool dependency information**

The module makefiles do not contain hard-wired paths for external third party tools but instead derive the information from the **SiteDefs** module. Though not precluded from doing so, YAM modules do not typically use autoconf. While autoconf is an excellent tool for "discovering" the capabilities of the environment and adapting the build accordingly, this works against the build homogeneity required to use link modules.

To provide this homogeneity, **YAM** tracks and version controls information about external software versions used during builds and at run-time. As a result developers can rollback the software - and the external tool configuration - to earlier points in time.

† **Support for multi-site development**

To support distributed teams that do not share a common development environment, and for supporting builds of software for deployment in other environment, **YAM** supports the definition of **site** specific build information in the **SiteDefs** module. This information is version controlled. link modules cannot be shared across different sites since the binaries may be incompatible. Developers can specify custom site information to override the information in the **SiteDefs** module. This is convenient for users working in isolated environments such as laptops or on subnets with different software environments.

† **Support for multi-target development**

Often, the development effort needs to support the developed software on more than one target platform hosts. A target platform here is used to denote variants in computing hardware, different types or flavors of operating systems, or differences in compilers for the target hosts. The key point is that binaries built for different targets cannot be inter-mixed. **YAM** provides build conventions and rules to support building modules for multiple platforms within the same directory structure. This is convenient for verifying and testing new capabilities across all the targets of interest in one place. **YAM** also provides convenient rules for building modules in parallel for the different targets. There is also support for gracefully disabling a module's builds for targets it does not support.

† **Module inter-dependency tracking**

YAM provides support for automatically extracting information about module inter-dependencies[2]. This information is very useful for generating alerts about modules impacted by API changes within a module, as well as for automatically pulling in modules needed by a module into a sandbox.

† **Software delivery and deployment**

A recently added **YAM** capability is the support for creating relocatable RPMs [12] for **YAM** packages and modules for external deployment of the software. An earlier capability for creating customized package tarballs for deployment continues to be supported.

† **Development effort is organized as a YAM Project**

A single **YAM** installation can be used to support multiple **YAM projects**. A **YAM** project denotes a development effort with a source repository, its own releases area, its own module/package definitions and its own releases database. **YAM** allows project-level as well as user-level tailoring of the default behavior of **YAM** commands to conform to policies defined for the development effort.

A. Summary of YAM Commands

YAM provides a small set of commands for developers to support the development work. The main ones are:

- **setup:** Creates a sandbox for a **YAM** package.
- **rebuild:** Updates work and link modules in a sandbox.
- **relink:** Updates link modules in a sandbox.
- **save:** Makes a module release.
- **sync:** Syncs up a module branch with new releases.
- **pkgnewrelease:** Makes a package release.
- **mkmodule:** Creates a new **YAM** module.
- **mkpackage:** Creates a new **YAM** package.
- **latest:** Returns latest release information for a module, package.

These commands support a number of options as well.

B. Customizing the Development Model

Some of the key factors that shape development policies include the size, stability and experience level of the development team, mission criticality and schedule for the software, homogeneity in the software, product-line needs, extent of software sharing, number of build targets and size of the codebase. **YAM** supports the tailoring of the development model to match a team's constraints, policies and needs. Examples of such fine-tuning include:

- Support for project and user specific customization of the default value for the **YAM** command options. For instance, the default behavior of the **setup** command is to create sandboxes with link and work modules on user specific branches off of the latest module releases. This default behavior can be changed to instead check out work modules on the main trunk so that branches are not created automatically. Another easy customization is to not use latest module releases (which represent the bleeding edge) by default but instead module releases from the latest package release baseline.
- Support for project specific customization of make file templates and rules.
- Support for disabling the use of link modules. Not defining a module release area for a **YAM** project completely disables the importing of link modules from the module release area. In this case, read-only versions of source code for the specified link modules are checked out and tagged.
- While use of parallel makes is supported for multi-target development, this can be disabled if so desired.

Tailoring such default behavior can have profound effects on a team's development patterns and dynamics.

III. Meeting Development Hierarchy Needs

In this section we examine how the elements of **YAM**'s software development framework described in Section II address the software development hierarchy of needs from Section I-B.

A. Meeting Basic Development Needs

YAM provides extensive support for meeting basic development needs including

- version control support using CVS
- module/package level release management, automated ChangeLog entry generation
- tracking of external software dependencies
- build management with make conventions for modules
- support for multi-target development

[2]Currently implemented for only C/C++ parts of the software.

YAM does not directly provide support for bug-tracking, test harnesses or documentation generation - all of which are important requirements for software development. There are several good open-source and commercial options available to meet these needs. **YAM** does allow make rule and other interfaces to such infrastructure tools.

B. Meeting Stable Concurrent Development Needs

† **Support for independent development sandboxes**
One of **YAM**'s central tenets - supported by the toolkit - is the use of independent sandboxes by developers for pursuing their development activities without stepping on one another's' toes.

† **Concurrent development on short-lived branches**
Software changes are in many ways like seismic tremors. A long sequence of mild tremors is easily handled, while large shocks can be very disruptive. Frequent branch/release cycles allows the team to incrementally absorb and adjust to each other's changes with relative ease. The use of private CVS branches allows isolation of developmental, immature code to avoid premature interactions and instability. Furthermore, it allows multiple developers to be work independently on the same module with minimal coordination overhead. Developer's merge only mature and tested code into the release branch. **YAM** provides commands that simplify the creation and merging of module branches. This helps keep branches short-lived avoiding integration difficulties and instabilities.

YAM uses conventions for auto-generating strings used as release and branch tags. Release tags contain a release number that increments with every release, and branch tags contain the release tag as well as the user name for private branches. To allow catching up with released changes for a module, **YAM** provides a **sync** command that creates a new branch by merging in development on an existing branch with the released changes on the main trunk.

† **Isolation from others' releases**
During development, modules continue to evolve and be released by the team. However, individual developers are free to choose the right time to sync up his/her sandbox with the new releases. This choice helps avoid unexpected and forced sync up efforts that distract from the development at hand. If a release happens to contain a relevant bug fix or new feature then a developer may choose to sync up, or else he/she may well choose to wait till the development task is completed before bothering to sync up. Such measured isolation provided by sandboxes enables productive development even in the midst of fast evolving code.

† **Module-level encapsulation increases stability**
Requiring a module to work as both work and link modules has the important benefit of requiring an explicit definition of the externally visible API for the module. A powerful side-benefit is that it enforces **module-level encapsulation** on the module's public interface, which in turn helps significantly reduce the coupling across modules. While compilers enforce class-level encapsulation by disallowing access to private and protected parts of a class, the module-level encapsulation means that modules cannot access classes, header files or libraries from other modules that have not been exported by the module. This mechanism allows the enforcement of private and public parts of a module and the permissible cross-coupling across modules. The encapsulation implies that even significant changes within a module that do not effect the exported API remain hidden from other modules. Such compartmentalization of changes increases system level stability.

C. Meeting High Quality Software Needs

† **module-level encapsulation improves organization**
module-level API encapsulation enforces clear functional and interface boundaries across modules reducing the possibility of hidden coupling and interactions across the modules. The only coupling possible across modules is that through the exported API from the modules. Thus classes, header files, libraries etc. that are not explicitly exported remain private to the module. This encapsulation takes the compiler enforced encapsulation of class definitions much further by enforcing privacy of classes themselves! Exposure of classes within the module API has to be explicit - avoiding inadvertent inter-module cross-coupling and complexity in the process. It has been observed [1] that such information hiding is the key to improving software quality. The encapsulation also exposes poor module definitions and improves software organization. Poor module definitions are evident by the large number of links that need to be exported helping improve software organization and design transparency.

† **Separation of stable and immature software**
The module-based organization is especially conducive to keeping stable and legacy software in modules different from those with new and immature software, allowing careful monitoring and attention to the new developments.

† **Refactoring of software is not a big deal**
A key aspect for improving software quality is the ability to refactor as the need arises. The ability to do so also reflects good API design. The module-based organization together with the minimal API exposure facilitates refactoring by localizing changes to modules, reduces the risk and impact on the rest of the system, and helps reduce the effort needed to refactor it. One strategy, when significant refactoring is required, is to create a new module that co-exists with the existing one. This facilitates side-by-side testing, and on conclusion, the old module is retired and replaced with the re-factored one.

† **Extensive module verification prior to release**
There is a strong requirement for developers to throughly test their changes before releasing them to the rest of the team. Furthermore, **YAM**'s release commands automatically run several checks to verify a module's readiness for release such as: that the modules are built against the latest link module; that all binaries are properly built; and that a ChangeLog entry has been created before allowing a release to proceed. Such continual testing of incremental changes and monitoring of quality helps improve the overall quality of the software.

† **Continual integration and test**
When there is a miscue, rolling back a change does not require a big effort either. The team settles into a "release-early, release-often" process where the software evolves and anneals at a steady and smooth pace. As changes keep merging into the

release stream, they continue to be exercised and shaken-out as the team uses them. These continual development cycles painlessly replace the carefully coordinated integration efforts that are otherwise needed to merge changes from long-lasting branches. **YAM** makes co-developers into the first line of beta testers of new changes. This takes advantage of the observations in [2] that "all debugging is parallelizable" and "given enough eyes, all bugs are shallow". **YAM** embeds continual quality improvement into the development process itself reducing the effort needed for addition quality assurance efforts.

When others on the team make module releases, it takes only a short duration for a developer to update their sandbox to pick up the new release as a link module in their sandbox since no builds are involved. This enables continual exposure and use of new developments by the team making them effectively beta testers on an ongoing basis. This also reduces the impact on the system from each of these smaller releases facilitating a smooth annealing of the ongoing software development.

† **Support for sub-system packages**

While **YAM** packages are used for product configurations, they can also be used to define sub-system configurations. Thus teams responsible for sub-system development can manage their development efforts using the same **YAM** infrastructure as for the system-level development. Working with sub-system packages allows the teams to develop, test and manage releases at the sub-system level. This facilitates a natural partitioning and decentralization of the development effort, enabling thorough vetting of the sub-system components before integration into the rest of the system.

† **Support for developmental configurations**

The process of software development often requires the creation of development configurations such as for unit testing, with instrumentation for debugging and profiling, stub software, demos, training, system level tests. The support software needed for such configurations is valuable for the development effort. The **YAM** framework insists that **all** software - including such support software - be captured in **YAM** modules. Furthermore, **YAM** allows the creation of **YAM** packages for these important development configurations. Sandboxes for these packages are easy to create, and their evolution can be tracked through package releases. Thus, **YAM** brings such support software and development configurations out of the hidden shadows and integrates them directly into the mainstream development effort. Once available, it is not surprising to see how often such configurations get used. Furthermore, being available as modules, portions of such support software get adapted for other development configurations.

† **Module level regression tests**

The module level organization also facilitates the use of regression tests throughout the software by including module level unit tests within the module. The make system supports rules for automating the running of regression tests in a uniform manner across the modules. Developers are expected to run and pass the regression tests before making a module release. This allows version control of the test code and its evolution with changes in a modules.

† **Package level regression tests**

Since packages represent runnable applications, system-level tests for the packages are needed and are hosted and version controlled within modules that are part of the package.

† **Built-in reuse model improves quality**

The very process of making software shareable by organizing them as modules promotes their reuse by other development efforts. Such reuse helps improve the quality of the module software. It promotes the "grass grows taller as it is grazed" [2] principle for software development.

† **Multi-site and multi-target support**

By building in support for multi-site, multi-platform development **YAM** avoids messy issues that can arise when porting software to a new target, or a new compiler needs to used, or the software needs to be deployed in a new environment. The transition effort to accommodate such events are easily handled by adding new targets and sites to the **YAM** project configuration. This requires no changes to the existing targets and sites and can thus be developed and tested independently without destabilizing and impacting ongoing development.

† **Templatized makefiles**

YAM provides makefile templates (which can be tailored by projects) as well as a host of helper make rules and conventions that allow the module level makefiles to be very simple. This allows the use of uniform conventions across the software, and also eliminates the burden of developing and maintaining complex make rules from individual modules.

† **Monitoring module API changes**

YAM supports the automatic generation of inter-dependencies among the C/C++ modules. With the decentralized organization of the software into modules, such dependency information is valuable for assessing the impact of API changes within a module on other modules prior to making the changes. Furthermore, the release process uses this information to alert the user to any API changes that may have occurred so that the user is able to follow up and make additional releases as needed.

D. Meeting Rapid Development Needs

† **Fast development sandbox setup and turnaround**

Link modules allows new sandboxes (even large ones) to be built up with scaffolding code easily and quickly. The fast sandbox setup also helps developers get used to the idea of creating and abandoning sandboxes for specific developments as a matter of course. Such agility in the development process is in sharp contrast to the not uncommon behavior where developers hang onto their functional sandboxes for as long as possible fearing long build times for new sandboxes, or from fear of crippling their existing sandboxes through a merge!

Furthermore, the separation of stable and developmental software into different modules allows sandboxes to use link modules for the stable parts of the software and work modules for the developmental modules.

† **Fast system sandbox setup**

Link and work modules also allow the fast set up of sandboxes for integration, test, training and demonstrations.

† **Handling large code-base**

As the code-base gets large, build and checkout times for the full source can become unacceptably large. However, the size of the code-base is typically a non-issue during day-to-day

development for the **YAM** framework. At any given time, an individual developer is typically working on a small number of work modules. The build time for a sandbox is directly proportional to the size of these modules alone since the rest of the modules serve as scaffolding link modules.

† **Parallel makes for multi-target builds**

As part of its support for multi-target development, **YAM** provides build and make rules to carry out builds for all the supported targets in parallel. Thus the build time stays more or less constant as new targets are added as opposed to scaling linearly with the number of targets.

† **Simplified branch and release process**

YAM provides scripts that make the creation of module branches effortless for the user. The user does not have to know the intricacies of the branch and merge process of the underlying version control system. **YAM** uses conventions for tags and branch names which once again the user does not have to deal with. The mechanics of the module development and release cycles take little effort. Such low cost encourages developers to release modules and terminate the branches with incremental developments and avoids the big integration push-ups that can result from long-lived branches.

† **Decentralized development and release process**

Having organized the software into modules with strong encapsulation allows **YAM** to decentralize the development process and reduce the mundane coordination overhead within the team. The compartmentalization of the software functionality, the isolation from developing on branches, and the incremental development process provide a very stable foundation to add new capabilities with reduced risk to the development process. Even when changes turn out to be not acceptable, rolling back is not a big effort. The reduced risk allows developers to spend their energy more productively on development rather than on coordination overhead. Appropriate team coordination however is still needed when far-reaching module API changes are to be undertaken,

† **Metrics collection from releases meta-data**

The releases database records information about the module and package releases. For those so inclined, the data is available to analyze and fine tune the development process. It is easy to query the database for information such as: modules that are going through a lot of change; or ones that are being worked on simultaneously by several developers; or assess the team's productivity in terms of the number of releases.

E. Meeting Reuse Needs

† **YAM module interfaces facilitate reuse**

YAM's module and package concept is at the heart of module reuse. While there are virtually no restrictions on what software can go into a module, the process of assigning a software to a **YAM** module requires a few critical steps that go a long way to making the module shareable. Firstly, the modules public interface has be explicitly defined so that it can be exported. Similarly, the dependency of the module on other modules, as well as external software, has to be clearly defined as well. These steps very clearly define the functional boundaries and interfaces for the module, protect from hidden dependencies and facilitating sharing of modules.

† **Packages can share modules**

YAM modules can belong to more than one **YAM** package. Packages are simply bundles of constituent modules, and there is nothing package specific within modules themselves. **YAM** provides simple mechanisms to add and remove modules from package definitions to facilitate the sharing of modules.

† **Standardized make interface for modules**

YAM's use of uniform conventions and rules for building a module makes it easy for users to bring unfamiliar modules into the mix without having to work through complex build issues that might otherwise be necessary. Thus it is trivial to reuse a module from one **YAM** package in another.

† **Packages are also reusable**

YAM allows packages to contain other packages. Such containment simply adds the modules from the contained package to the list of modules for the parent package. Such hierarchical package definitions are effective for not only managing a product-line, but for also building up more complex products from existing ones.

† **Package releases manage change evolution**

As modules evolve, there is no guarantee that an arbitrary combination of module releases are compatible with one another. Package releases establish baselines where the combination of module releases in a package release have been tested for compatibility. Regular package releases help the team to track the evolution of the package as well as to help bracket the source of any bugs that may be discovered. The existence of such baseline releases also establishes the quality of the constituent modules and facilitates their reuse elsewhere.

IV. USAGE

YAM started out as a framework for fast-paced simulation software development by the DARTS Lab team. **YAM** has evolved considerably over the years and has been adopted by several other teams for managing their software development. These development efforts include mission-critical software development for the Mars Exploration Rover flight software [13], Space Interferometry Mission [14] and the Deep Impact fault protection software [15] as well as R&D projects such as CLARAty robotics software development [16]. The size of development teams has been in the range of 3-30 developers. Most of these development efforts have involved collocated teams sharing common development environments permitting the use of link modules. Indeed, even though several of the teams are working towards the development of a single product, they have nevertheless adopted **YAM** to support their development.

It is noteworthy however that while using **YAM**, there are significant variations in the development policies and models used by the different teams. Some of the determining factors in defining the development policies include the stability of the development team, experience level of the team, mission criticality and schedule, homogeneity in the software and product-line needs. Some teams chose to not introduce the use of the novel concept of link modules to the team until the team gained sufficient familiarity with the **YAM** development process. Other teams have chosen to assign ownership of modules to sub-teams or individuals. Yet others allow package releases by

only a single individual responsible for creating baseline builds on a regular basis. Some teams do not use branches and carry out all development on the main trunk. **YAM** provides mechanisms to tailor project configurations to meet the policy needs of individual teams.

One of the issues teams need to plan for is the need to provide adequate training and mentoring to help new team members familiarize themselves with the range of new concepts such as link/work modules, packages, sites, targets etc. that come with **YAM**. Of course, such training would be necessary for any development process in place.

At the DARTS Lab, the **YAM**-based development supports a product-line consisting of physics-based simulation application products for a range of application domains including spacecraft, planetary rovers, entry/descent/landing systems and airships. These distinct simulation products share a number of infrastructure modules such as for vehicle dynamics, device and environment models, simulation framework, graphics visualization, and user interfaces. It's **YAM** setup has around 200 active **YAM** modules and 30 **YAM** packages and the software supports 2 to 5 build targets. The development team is collocated and has ranged in size between 6 to 10 developers and includes engineers and software personnel. The software is a mix of C/C++ software, with significant amounts of Python, Tcl, Perl and some Fortran and is approximately 2 million lines of code. All development takes place on branches and the team makes extensive use of link modules. The development process is fairly decentralized within the team. The development pace is fast with over 2400 module/package releases during 2004, and over 2800 in 2005. Of course, the **YAM** software is itself managed as a **YAM** module!

V. Conclusions

The **YAM** software development framework provides solutions to many thorny challenges involved in team-based concurrent software development, managing software product-lines and sharing software. We have described the ideas and principles that form the basis of **YAM** and discussed how they address the hierarchy of needs for effective software development. The **YAM** framework integrates concepts spanning software organization, build management, release management, and software reuse to provide a nimble development processes for managing complex software development. Contrary to the belief that such rapid development processes are not suitable for mission-critical software development, **YAM** is being successfully used for such development by a number of NASA missions.

To date, **YAM** has largely been used by mid-sized, collocated development teams. However, **YAM**'s multi-site development support can be adapted to large-scale developments that are organized around several mid-sized teams. Open source projects can benefit from **YAM**'s support of multi-site development, module-based software organization, bazaar style decentralized development and smooth branch and release cycles. However link modules are likely to be less useful for open source projects since developers are not collocated, and do not share common development environments.

YAM has benefited greatly from other open source software and our goal is to make it publicly available in the near future.

Acknowledgments

We would like to thank Garth Watney for the many constructive suggestions that have helped improve **YAM**'s usability over the years. The research described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration, and has been partly supported by the National Science Foundation Grant ASC 92 19368.

References

[1] F. N. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 1995.
[2] "The Cathedral and the Bazaar website." `http://www.catb.org/~esr/writings/cathedral-bazaar/`.
[3] "Agile Alliance website." `http://www.agilealliance.org/`.
[4] "Extreme Programming Resource website." `http://www.xprogramming.com/`.
[5] "YaM website." `http://dartslab.jpl.nasa.gov/yam/`.
[6] "DARTS Lab website." `http://dartslab.jpl.nasa.gov/`.
[7] A. Maaslow, *The Farther Reaches of Human Nature*. Viking, 1971.
[8] "CVS Wiki website." `http://ximbiot.com/cvs/wiki/index.php?title=Main_Page/`.
[9] "MySQL website." `http://www.mysql.com/`.
[10] "PHP website." `http://www.php.net/`.
[11] "The Perl Directory website." `http://www.perl.org/`.
[12] "RPM Package Manager website." `http://www.rpm.org/`.
[13] "2003 Mars Exploration Rovers website." `http://www.jpl.nasa.gov/missions/current/marsexplorationrovers.html/`.
[14] "Space Interferometry Mission website." `hhttp://planetquest.jpl.nasa.gov/SIM/sim_index.cfm/`.
[15] "Deep Impact website." `http://deepimpact.jpl.nasa.gov/home/index.html/`.
[16] "CLARty website." `http://claraty.jpl.nasa.gov/`.