The Software Development Process of FLASH, a Multiphysics Simulation Code

Anshu Dubey∗, Katie Antypas†, Alan Calder‡, Bruce Fryxell§, Don Lamb∗, Paul Ricker∗, Lynn Reid†, Katherine Riley∗∗, Robert Rosner∗, Andrew Siegel∗∗∗, Francis Timmes††, Natalia Vladimirova‡‡, Klaus Weide∗

∗ University of Chicago, Chicago, IL 60637, USA
† Lawrence Berkeley National Laboratory, Berkeley, CA, 94720, USA
‡ Stony Brook University, Stony Brook, NY, 11794, USA
§ University of Michigan, Ann Arbor, MI 48109, USA
¶ University of Illinois, Urbana, IL, 61801, USA
∥ University of Western Australia, Crawley WA 6009, Australia
∗∗ Argonne National Laboratory, Argonne, IL, 60439, USA
†† Arizona State University, Tempe, AZ, 85287, USA
‡‡ University of New Mexico, Albuquerque, NM, 87131, USA

Abstract—The FLASH code has evolved into a modular and extensible scientific simulation software system over the decade of its existence. During this time it has been cumulatively used by over a thousand researchers in several scientific communities (i.e. astrophysics, cosmology, high-energy density physics, turbulence, fluid-structure interactions) to obtain results for research. The code started its life as an amalgamation of two already existing software packages and sections of other codes developed independently by various participating members of the team for other purposes. In the evolution process it has undergone four major revisions, three of which involved a significant architectural advancement. A corresponding evolution of the software process and policies for maintenance occurred simultaneously. The code is currently in its 4.x release with a substantial user community. Recently there has been an upsurge in the contributions by external users; some provide significant new capability. This paper outlines the software development and evolution processes that have contributed to the success of the FLASH code.

Index Terms—FLASH code, community code, software evolution, adaptive mesh, case study

I. INTRODUCTION

The FLASH code, developed at the Flash Center at the University of Chicago, has evolved into a well-architected software system with a wide user base. The FLASH code has been applied to a variety of astrophysics problems including supernovae, X-ray bursts, galaxy clusters, and stellar structure, and also to more fundamental problems such as fluid instabilities and turbulence. Recently the code has also been used to study high-energy-density environments and fluid-structure interactions. Majority of FLASH applications involve compressible reactive flows with strong shocks. Additionally, the range of temporal and physical scales in the problems of interest is such that all scales cannot be fully resolved. The simulations, therefore, rely not only upon adaptive mesh refinement (AMR), but also on models of some physical phenomena for which direct numerical simulation is not feasible. The current release of the FLASH code, which is used for simulations of Type Ia supernovae and high-energy density physics (HEDP) in the Center, includes necessary capabilities such as compressible hydrodynamics and magnetohydrodynamics solvers, nuclear burning, various forms of equations of state, radiation, laser drive and many more.

Even though publicly available astrophysical simulation codes such as the Barnes and Hut tree code [1], and ZEUS-2/3D [2] had been in existence and enjoyed fairly extensive user bases, the concept of a community code was fairly new to the astrophysics academic community at the time of the original FLASH release. Over the last decade many more astrophysical codes such as Athena [3], ZEUS-MP [4], Gadget [5], and Enzo [6] have become available. One of the key reasons for the success of FLASH was the long term development horizon due to the multi-year funding provided by the ASC program, which allowed for an unprecedented commitment to code support. While other funding programs such as NASA’s HPCC had supported software development, the only public releases from those codes were in the form of making them available through some repositories without any support for the users.

This paper outlines the design decisions, software development processes, and interdisciplinary interactions that have contributed to the success of the FLASH code. The remainder of the paper is organized as follows: Section II describes the approach during the early versions of the code, Section III outlines the evolution of code development and maintenance practice in the later versions of the code. Section IV outlines the process used for transitioning between major versions of the code. The user community is described in Section V, and
Section VI describes external contributions to the code base. Section VII summarizes the lessons learned during the evolution of FLASH that are likely to be applicable to development of other multiphysics scientific codes and their adoption by their target research communities. Finally, in Section VIII we state our conclusions.

II. EARLY VERSIONS

The Flash project started as part of the Department of Energy University Alliance program, under its Advanced Scientific Computing Initiative (ASCI, now ASC) program; the aim of this program has been to explore the challenges associated with physics problems that are complex and computationally demanding, but for which the possibility of experimental validation is difficult, if not impossible. The specific aim of the Flash project was to study astrophysical thermonuclear explosions, and involved a broad range of scientists, from astrophysicists and physicists to applied mathematicians and computer scientists.

The key initial question facing our collaboration was how to proceed? This question needed to be confronted right from the outset because the astrophysicists viewed the challenge quite differently from the more computer science-oriented collaborators. The astrophysicists viewed the problem of building the FLASH code as an iterative process, one in which the starting point was a functioning code that was already capable of addressing at least a few aspects of the target problems. In contrast, the computer science-oriented collaborators viewed the problem from the perspective of building a formal modular code structure (e.g., a “framework”) that was sufficiently general that all the expected subsequent modules needed to address the Center’s target astrophysics problems could be accommodated. During the initial stages of the project, we pursued both paths, in the expectation that the iterative path would suffice to satisfy the astrophysicists, while the computer science-oriented path was being developed. The motivation for pursuing this double-pronged path was the notion that formalization of the modular structure of what was to become the FLASH code right from the start would result in a better-performing and better-functioning code – together with the need to keep the application scientists (i.e., the astrophysicists and physicists) engaged by giving them a tool that allowed them to start down the path of addressing the target nuclear astrophysics problems.

However, it soon became increasingly clear that the available resources did not suffice to fully support both paths; and as a result, internal conflicts between the two “camps” became increasingly vigorous. This conflict, and the fact that the computer science-oriented path was relatively slow to produce code that engaged the application scientists, led to the decision to follow only the iterative path. The same constraints did not apply to the later versions because a version of production ready code was always available to the scientists. Therefore, it became feasible to move away from a purely iterative path and to devote some resources to building a modular framework for the code in parallel to supporting the production version. The iterative path in the early stages proved to be beneficial in another way; the code developers had become familiar with all the code components, and therefore the architecture that emerged in the later versions was better suited to the demands of the various code components.

The aforementioned iterative path started with an amalgamation of two already existing software packages, PARAMESH [7] for AMR, and PROMETHEUS [8] for hydrodynamics, each offering a subset of capabilities required for the numerical simulations being undertaken at the Center. In particular PROMETHEUS was selected as the hydrodynamics solver because it was based upon the PPM scheme which provided compatibility with flexible implementations of equations of state [9]. Other needed capabilities were added from sections of several other codes that had been developed individually by various members of the Center in their earlier research, such as the equation of state for degenerate matter and reaction networks for nuclear burning [10]. Several design decisions made in the early days of the code were forced upon the developers by the constraint that the code be usable in production mode within months of conception. The other equally influential factor was the diversity of the sources of code components; they had been developed in different research groups with diverse priorities and different coding practices. Together, these two factors dictated that the evolution of code architecture and capabilities be incremental, with lessons learned and absorbed at each stage.

The key components of FLASH, being isolated codes, were written using coding practices typical in the Fortran 77 era. The variable names were limited to six characters, the functions lacked interfaces, the data exchange was implemented through common blocks, and arrays were declared statically. Since the code was expected to handle a variety of physical situations, it was necessary to be able to selectively include different components of the code, requiring different physical variables. The selective functionality of the code was achieved by creating an “inheriting directory structure” described in [11; 12], and the introduction of a configuration mechanism, the “setup tool”. Along with the setup tool, written in shell script, we introduced the concept of “Config” files. Config files are text files in FLASH-specific syntax, which contain all the meta-information needed by the setup tool to implement the architecture. The setup tool also brought object-oriented concepts such as inheritance and encapsulation to FLASH. Subsequent enhancements in the FLASH architecture have been achieved by augmenting the Config file syntax and extending the capabilities of the setup tool.

We introduced version control from the outset: CVS [13] up to FLASH2, and SVN [14] from FLASH3. There was a concept-testing version, FLASH0, which was not released. The intent was to have a public release from the next version, FLASH1, by which time the code developers expected to achieve reasonable confidence in the correctness of the code. FLASH1.6 was the first version of the code to have a public release. Several application setups were developed during this phase which could be used to regression test the software.
Some tests also verified the computed solution against known analytical solutions; these tests later formed the basis for the first iteration of the unit test framework described in [12].

III. SOFTWARE PROCESS EVOLUTION IN LATER VERSIONS

A. Centralizing Data: FLASH2

FLASH already had many users outside the Flash Center when the need to further clarify and formalize its architecture was identified. The first goal in transitioning to FLASH2 was to remove the Fortran common blocks which make it extremely difficult to differentiate between variables being used in a function from those that are not. The variables were restructured for FLASH2 in three stages. First, common blocks were replaced by Fortran 90 modules with unrestricted use module commands in the calling routine. Then the use module statements were restricted to variables actually used within the calling routine. Finally, direct access to module variables was replaced by interface calls to a variable database. The variable database (a collection of F90 data modules) acted as a central repository of data being used in the simulation. While a centralized database proved to be a less than ideal solution, overall the transition was beneficial: the data was inventoried and the function of different variables was clarified, and global variables were separated from internal and auxiliary variables. What the centralized database failed to provide was a clarification of which unit owned a specific data item and which units could access and/or modify it. Without the ability to control data access, module encapsulation, and therefore extensibility, could not be fully achieved.

During this phase the process of code verification was also formalized. Though regression tests had been developed earlier for code verification, they were applied individually by the developers as needed. The tests were put together in a test-suite which was then setup to run nightly on multiple platforms for daily code verification. This practice has proven to be so beneficial to our confidence in the code and early bug detection that it has continued without interruption since then.

B. Decentralizing Data, FLASH3

While FLASH2 had successfully unraveled the structure of the code and module interdependencies by centralizing data management, the ownership and modifiability of data remained unresolved. Because all units interacted with the database using identical interfaces, the level of access granted to the units could not be differentiated for any given data item. A potential undesirable side effect of such undifferentiated access was that write permission on a data item made it alterable by any module, including those that shouldn’t have been modifying it. Additionally, lack of clarification about data ownership not only interfered with encapsulation of modules, but also created a high barrier to entry for new modules and new developers. Furthermore, the necessity of including the variable database in every setup precluded the possibility of devising unit tests for even the simplest of solvers. These disadvantages significantly compromised the extensibility of the code. Decentralizing the database, so that individual units owned and arbitrated on the accessibility of their own data, was the first major step towards realizing the true extensibility potential of FLASH.

Figure 1 shows data management in the three major versions of FLASH. In FLASH1, the data items are not explicitly owned by any single unit of FLASH, while in FLASH2, they are all owned by the central database. Neither version had the capability to arbitrate on which units can modify data, and when. In contrast, every data item except global constants is exclusively owned by exactly one unit in FLASH3. The owner unit defines the scope of the data within the unit, and its accessibility from outside the unit. Since much of FLASH’s coding standard is not programming language enforced, we have a script that runs nightly and flags all coding standard violations.

IV. TRANSITIONING BETWEEN VERSIONS

In transitioning between major versions of the code, there is a conflict between the need to have a robust code that can be used for running simulations with reproducible results, and the freedom to alter large sections of the code as needed for the new version’s coding standards and architecture. The transitions played out very differently between going from FLASH1 to FLASH2 and going from FLASH2 to FLASH3.

A. FLASH1 to FLASH2

The approach taken during the transition from FLASH1.6 to FLASH2 was to keep the functionality of the development and production branches consistent while focusing on fundamental changes in the way data was accessed by components of the code. This constraint of keeping the two branches synchronized appeared to be necessary at the time because we were still following the iterative approach to architecting the code. Also, the addition of new solvers did not stop during the architecture transition, and new solvers were primarily added to FLASH1.6. However, we discovered that the synchronization added unnecessary complexity and sometimes significant amount of code duplication in many parts of the code. From the user’s perspective the transition was relatively transparent and smooth. However, the developers’ task became extremely difficult with significant loss of developers’ time spent in maintaining both branches. Even worse, the learning curve for the new developers became too steep. In the end, these experiences proved to be an important lesson learned, and we moved away from the purely iterative approach in the transition from FLASH2 to FLASH3.

B. FLASH2 to FLASH3

During the transition from FLASH2 to FLASH3 we maintained two different repositories. The transition methodology was to build the infrastructure of the new version of FLASH around the mesh package and the IO library and then confront the solvers. Here, because the changes were deep, the interfaces in the wrapper layers were designed to be fundamentally different without concerns of backward compatibility. Thus the backbone of FLASH3 was built and tested almost in isolation,
and a unit test framework was built alongside. During this phase new solvers were added exclusively to FLASH2.

The migration of solvers to the new version started with the most mature solvers first because they weren’t undergoing continuous development. As a policy, once a solver was designated to be in the transition phase, it was frozen in FLASH2; any changes to the solver had to be made in collaboration with the group implementing the transition to FLASH3. Since we were focusing on mature solvers, there weren’t many instances of such developments. Next we transitioned all the applications (the corresponding initial/boundary conditions and any custom capabilities implemented for the application) that exclusively used the already migrated mature solvers and started the verification process with nightly regression testing. From this point on the new solvers were added directly to FLASH3.

Thus FLASH3 was already fairly mature and robust when we tackled the most challenging part of the transition; namely the newer, more frequently modified code units. For this stage, we chose to concentrate on moving one application at a time; that is, focus on transitioning all the units in use by a specific application, rather than picking the units at random, or based upon some other criterion such as complexity. The rationale for application specific approach was to minimize the disruption time for specific research in the Center by concentrating all development resources to it. It was easier to get co-operation from the scientists if the transition time was short, and they received a more robust and better supported code in exchange. Once the first research application became available in FLASH3, the remainder of the transition was quite rapid and smooth. It was helped considerably by the active participation of astrophysicists in the transition.

C. Interaction Between Domain Science and Code Development

From the beginning there has been an interesting interplay between domain science and code development at the Flash Center. Most developers of FLASH1 were the scientists who wanted to use the code for their own simulations. However, and this has proved to be the most far reaching impact on FLASH development, several of them were also committed to the cause of open science and wanted the code to be public. That desire is why certain farsighted architectural features such as inheriting directory structure were introduced in FLASH1 that continue to exist in the most recent version. The demography of the developers changed between FLASH1 and FLASH2 developments. The architecture restructuring that resulted in FLASH2 was dominated by code writers more interested in the software development and maintenance than in using the code for their own research. This change of personnel led to an inadvertent bifurcation between the astrophysicists and programmers, causing a large delay in adoption of FLASH2 for production. Astrophysicists, being more comfortable with FLASH1, found it more familiar to customize for their needs. Added to that, there was a performance loss in FLASH2, partly because of data centralization, and partly because we could not use more advanced compiler optimizations on some of the major production platforms and still obtain reliable results.

In contrast, FLASH3 began to be used for production within the Flash Center as soon as minimum required capabilities were brought over into the new version. There were several reasons for this rapid change. The most important reason for early adoption of FLASH3 was that it addressed a large body of reported bugs, limitations and requested features that had been accumulated due to users exercising FLASH2, making FLASH3 far more robust and accurate. However, the experience of FLASH1 to FLASH2 transition also played a

<table>
<thead>
<tr>
<th>FLASH 1.6</th>
<th>FLASH 2.0</th>
<th>FLASH 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data residing in common blocks</td>
<td>Data residing in central database</td>
<td>Data owned by individual units</td>
</tr>
<tr>
<td>No explicit ownership of the data</td>
<td>Database owns all data</td>
<td>A unit can use its own data with a “use” statement</td>
</tr>
<tr>
<td>Any unit wanting to use can include a common block</td>
<td>Any unit wanting to use must query database</td>
<td>Data belonging to other units accessed or modified only through API calls</td>
</tr>
<tr>
<td>No way to prevent unauthorized modification of data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. The sharing of data items in three different versions of FLASH. The example is from the Driver module/unit.
role. The Center leadership had realized the importance of cooperation among various groups and took extra care to foster inter-group interactions. This led to greater trust between the scientists and code developers. Once the basic infrastructure of FLASH3 was ready, the code developers and local users cooperatively prioritized the code units to be transported. As a result, the alpha version of FLASH3 was already in production in all of Flash Center’s simulations long before it was publicly released.

V. USER COMMUNITY

The FLASH Code has attracted a wide range of users and has become a community code primarily in, but not limited to, the astrophysics community. Feedback from the user community indicates the degree of involvement of FLASH users. We conducted user surveys in 2005, 2007, and 2009. In each case, the target audience for the survey was the user community registered in the external FLASH-USERS email list.

Over three hundred people responded to the first two surveys. Most users at the time were from the United States with an academic affiliation, but there was a large geographic spread of users and a diverse range of companies and research institutions represented. Usage of the code fell into three categories: as a primary research tool, for verification and validation, and as a sample code for educational purposes. Principal research areas reported by the users were in astrophysics, but a significant number of users reported studying computational fluid dynamics or working in algorithm development. Users preferred using FLASH for both the code capabilities and the extensive support of good documentation, ease of use, and widespread users. Further concrete details of the first two surveys are provided in [12].

The survey conducted in 2009 aimed to compile a database of all authors who were on a published paper that utilized the FLASH code. This compilation represented a different way of determining FLASH users, and included users who benefitted from research produced with FLASH, but may not have been directly involved in running the code. At the time of the survey, over 600 people were collaborators on research utilizing some version of the FLASH code. Only 81 of those people were directly employed within the ASC Flash Center.

As of 2012, the Flash Center hosts a web-based database [15] of published papers that have used the FLASH code to obtain some or all of their results, or are about the code. We keep the database updated through a Bibliographic review of newly published papers conducted on a regular basis. Users are also encouraged to directly upload new publications into the database via a web interface. Figure 2 shows the breakdown of publications by the year, starting from 2000 when results using FLASH first started to appear in the literature. At the time of this writing the database contains more than 800 publications with a combined authorship of more than 1100.

VI. EXTERNAL CONTRIBUTORS

In recent years there has been a noticeable growth in code contributions by the external FLASH users. These code contributions occur in three major ways: a Flash Center member reaching out for collaboration and sometimes getting some code development in return; or a user reaching out to the FLASH team with a specific goal, often resulting in code contribution; or users may develop some added capability for their own research, which they later want to share with the community. In such instances the Flash Center has traditionally encouraged and supported visits by potential code contributors to facilitate the process. The contributions have ranged from astrophysical and other general-purpose solvers to infrastructure enhancements.

It has often been necessary to have some involvement from the core FLASH developers to facilitate any external contribution into the code. Since the release of FLASH3.0, such involvement can be minimized to the administrative task of inclusion into the code repository and regression test suite. Additionally, there is a new mechanism in place which can potentially eliminate even this constraint. The FLASH distribution now has the provision for add-on units, which are code segments that reside outside of the official distribution, but can work seamlessly with the code. Some of these code segments may be available from the Flash Center website; these typically include the units that are available as-is, without any rigorous verification by the FLASH team. Some other code segments are available from their developers within and outside the Center who are listed in the release notes of the distribution. The listed persons can be contacted directly by the interested parties to obtain the code at the discretion of the developer. It is hoped that in the future this direction will eventually lead to a more public hosting of various code segments, with the ultimate goal of the embracing the open source model.

VII. LESSONS LEARNED

FLASH started out with legacy codes and took three iterations of code reorganization to achieve modularity and extensibility. Each iteration added to the knowledge about do’s and don’ts of an academic software project with ambitious scientific capabilities goals. One of the biggest contributors to the success of the FLASH code, and also for the research in the Flash Center using the code, is good communication between domain scientists and code developers that took time and effort to achieve. In an ideal situation the domain science would dictate the priorities and a rough timeline of code capability development, while the code team would be guided by these constraints within the framework of its development schedule without being forced to compromise on the software engineering and quality control. While we cannot claim to have achieved the ideal situation, we have strived for this balance, and therefore have a better software product than we would otherwise have. We rarely have feature creep, and a broad cooperation results in more rigorous verification of the code all through the lifecycle of a unit development. Some of the other important lessons learned are summarized below.
A. Public Releases

Early and frequent public releases of the code are greatly beneficial to the overall code quality. The general tendency among the code development projects is to unnecessarily delay making the code available to users. This hesitancy could be driven by the fear of embarrassment if the code is found to have faults, or it could be because of the often erroneous perception of having an advantage over the scientific competition. While releasing the code too early or too frequently is clearly undesirable, a judiciously chosen schedule for early and relatively frequent releases has two significant advantages: coding standards are followed more carefully and code verification is more rigorous when the code is exposed to external scrutiny, and the code is exercised in many different ways by diverse users and bugs are found and fixed more quickly.

B. Transient Developer Population

In academic settings many code developers are post-doctoral fellows and graduate students. This presents two challenges: priorities for the solvers and other capabilities change as individuals leave, and the expertise sometimes leaves with people making it difficult to maintain the corresponding code section. In the matter of code infrastructure it is critical to include software engineers in the team and have a lead architect to provide continuity. For physics solver units the most effective solution to the transient developer population is documentation. In addition to standard documentation such as a user’s guide and other online resources targeted at the users, the FLASH developers are encouraged to add extensive inline documentation in any code section that has even moderately complex logic. There is broad enough general technical knowledge among the Center members that someone can maintain a well-documented code section even if it isn’t in their own area of interest. A team with a breadth of knowledge and expertise is, therefore, also critical.

C. Backward Compatibility

Our experience has been that backward compatibility is not a very desirable goal during major version changes. We maintained backward compatibility in the function interfaces when transitioning from FLASH1 to FLASH2. It was expensive in terms of developer time and did not provide any tangible benefits. It did not accelerate the adoption of the newer version by the scientists, made the code chunky, and prevented some deep changes which would have allowed for greater flexibility. FLASH3 did not attempt backward compatibility, and focussed on getting the infrastructure right. This version was not only able to incorporate deep changes as needed, but was also adopted more quickly for production because it was noticeably more extensible and robust than version 2. We believe the reason why backward compatibility worked so differently for FLASH compared to commercial software is that the primary use of the code is for fundamental research. For FLASH users having the state-of-the-art ideas incorporated into their tools outweighs the disadvantage of having to revise their customizations in the code, which because of the FLASH architecture tend to be isolated and contained in the Simulations unit.

D. User Support

Having a well defined user support policy is extremely important in convincing the community to use the code. While it may take significant resources initially to respond to all user queries, in time that effort reduces because the community becomes self-supporting. Providing comprehensive documentation which is easy to access and reference also helps reduce the demands for user support. One of the features of FLASH that has proved to be very popular among users is the quick start guide, and a collection of well documented example.
application setups. Many users report starting with an example setup and being able to quickly customize it for their own application, which considerably reduces their initial effort. Clear policies for contributions from the users and acknowledging those contributions is also extremely important.

E. Code Infrastructure

Having gone through three iterations of code infrastructure development, our finding is that the least cluttered code architecture provides the greatest flexibility and longevity. While it may seem like stating the obvious, scientific codes very often fail to achieve this elegance for several reasons. The most common reason is feature creep: almost all codes have features that seemed desirable at some point but either did not prove to be useful, or outlived their usefulness. Without regular pruning they continue to clutter the architecture. Another reason is that many scientific code development efforts consider the code infrastructure to be of secondary importance and devote the bulk of their resources to capability developments. This focus is especially true when development and resource allocation priorities are driven by near term scientific goals. The most important reason, however, is that it is extremely hard to achieve architectural simplicity in a complex software with many moving parts. It requires substantial investment on the part of the developers to understand the requirements, the limitations, and the idiosyncrasies of the core solvers as they relate to the infrastructure. The best time to devise and formalize code architecture is after the knowledge about the core solvers has been thoroughly internalized, and that requires a willingness to redesign and rewrite large chunks of infrastructure code at least in the early iterations.

The FLASH team has faced all three of the challenges described above. The first and the third challenge we were able to overcome because of major architectural revisions between the first three versions. The transition from FLASH1 to FLASH2 enabled the code developers to acquire the knowledge of the solvers, which was then utilized in architecting FLASH3. And because FLASH3 was not trying to be backward compatible with FLASH2, it was possible to prune the redundant features. The challenge of scientific priorities getting in the way of robust code infrastructure is in general difficult to overcome. All incentives to the domain scientists go against appropriate resource allocation to code infrastructure design since good infrastructure does not directly help with publications or proposals. What many groups fail to appreciate is that a well designed code infrastructure (or framework) is the only way to enable future capability extensions. It was only after FLASH acquired a really robust infrastructural framework in version 3 that the code became truly extensible. That design has proven to be robust enough that the transition to version 4, which is HEDP capable code, required only minor revisions to the architecture. As mentioned earlier, the FLASH team was fortunate in having funding that provided for software development. Now that the domain scientists have reaped the benefits of a well designed infrastructure, the culture at the Center continues to emphasize resource allocations for it.

F. Testing

Code verification in general gets some attention from scientific code developers, but daily regression testing, a critical aspect of development, all too frequently does not. Many code developments efforts have either an individual or a very small group involved. Therefore, they do not see the need for the rigor of daily testing. Sometimes even larger teams don’t test on a regular basis. The Flash Center runs a collection of problem setups on a number of platforms every day that exercise individual code units, their alternative implementations, and their interoperability with other code units. Our experience has been that we need to closely monitor our daily test-suite to maintain the reliability of the code units and their interoperability. Developers often test their modifications on a few problem setups before checking them in. Sometimes those modifications affect other parts of the code in a very obscure way. Because every modification in the primary development branch is subject to a range of tests, there is much greater chance that such unintended effects are detected early and fixed. The amount of effort needed to fix such inadvertently introduced faults is higher the longer they stay undetected. For all of these reasons, all FLASH code developers and internal users rely heavily upon the test-suite.

VIII. CONCLUSIONS

FLASH has had four major versions releases, where first release was provided within two years of the start of the Center. This version also laid the foundation of the code architecture with modularity and extensibility in mind. Version 1 had only one release, 1.6, before transitioning to Version 2 in 2003. The final release of version 2 was 2.5, occurring in 2006. Version 3 was released in 2008, with the alpha and beta versions appearing in 2006 and 2007 respectively. FLASH3 went up to 3.3 release. FLASH4.0 was released in 2012, without significant architectural or infrastructure change, instead with capabilities added that targeted a new research community (HEDP).

The development of FLASH and its wide adoption have followed a unique path among academic scientific codes. Some code bases, such as the codes in the lattice QCD community have full support of their community. Some others such as Zeus, Athena, Enzo and Gadget have been developed with very focused applications and have become popular among their narrow targeted communities. There was skepticism of FLASH’s broad based approach, where it was feared that extra overheads from multiple applications support would make FLASH uncompetitive with more focused codes. However, FLASH has proven itself to be a very nimble code, which has not only continued to evolve in capabilities, but has also kept up with the evolution in software engineering and hardware architectures. An unorthodox code development team with emphasis on breadth of expertise, and sustained multi-year funding have both been crucial in the success story. The additions of HEDP capabilities to the code, and several recent contributions by our external users, are a further testament to
the validity of FLASH development strategy and management philosophy.

ACKNOWLEDGMENTS

The authors would like to thank Jonathan Dursi and Mike Zingale for their efforts in making the FLASH code publicly available, and for their contributions to the code. Additionally, the authors want to acknowledge the code contributions from all internal and external users. The evolution of the FLASH code described in this paper was in part supported by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago under grant B523820.

REFERENCES