Decomposing Verification by Features*

Kathi Fisler[1] and Shriram Krishnamurthi[2]

[1] Department of Computer Science, WPI, Worcester, MA, USA
kfisler@cs.wpi.edu
[2] Computer Science Department, Brown University, Providence, RI, USA
sk@cs.brown.edu

Abstract. Practical program verification techniques must align with the software development methodologies that produce the programs. Researchers from several corners of software engineering have proposed similar models of program development in which modules encapsulate units of end-user functionality known as *features*. These models ameliorate some difficulties with conventional modular verification, such as property decomposition, while creating others, by contradicting assumptions that underlie most modern program verification techniques. This paper motivates the decomposition of systems by features and provides an overview of the challenges this poses to verification.

A Notion of Software Development

For program verification to thrive, verification methodologies must align with software development methodologies. This means that verification tools should be able to handle program fragments of the style and granularity that programmers produce. In addition, the effort to verify a program increment should bear some reasonable ratio to the effort to develop that increment.

An important trend in software development poses particular challenges to conventional methods of program verification. Our understanding of this trend is inspired by the picture in Figure 1 which Michael Jackson used in his presentation at ESEC/FSE 2001 (following his acceptance of the SIGSOFT Outstanding Research Award).[3] The box at the lower-left might be grossly characterized as the province of programming languages, proceeding from specifications to programs that, we hope, properly implement those specifications. This is the realm of solutions, (in Jackson's words, the *solution space*). The box at the upper-right is the domain of requirements engineering: the collection of processes, many sociological (and imprecise!), that glean requirements for a system from its users and other stakeholders or, more broadly, from the fuzzy blob that is the "real world". In Jackson's terminology, the transactions in this world must remain in the *problem space*.

* This work is partially funded by NSF grants CCR-0305834, CCR-0132659, CCR-0447509 and CCR-0305950.

[3] We have transcribed this picture from our notes; a related version is in a paper [1].

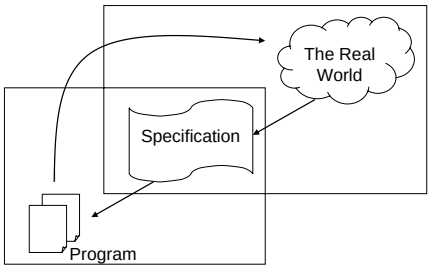

Fig. 1. The requirements-program feedback loop.

As soon as the program comes into existence, however, it itself becomes a part of the world. This invariably triggers a possibly new set of requirements. (As most requirements engineers and user interface designers will attest, a common user reaction is, "Oh, it does *that*?" followed by, "That is not what I meant at all. That is not it, at all" [2].) This cyclic dependency and directed flow of requirements is a source of many contemporary software development problems. This suggests that our techniques for software development should account for this by finding ways to be more pliable in the face of requirements changes.

Program Development Styles and Verification

These ideas have significant implications for program verification. Verification research has often assumed a simplistic model of program development, in which the verifier has a complete program to analyze against established requirements. The body of work on modular verification relaxed that assumption to handle portions of programs that correspond to units of separate compilation [3]. This position paper argues that emerging forms of software development embody different assumptions from most current verification methodologies. It is therefore essential for verification to support these development techniques; better still would be if verification could *exploit* it.

Some researchers [4] believe verification should be organized around *software components*. Unfortunately, the term "component" seems to mean too many things (and often too little) in the literature. In particular, a component may or may not have a direct relationship to the requirements that inspired the program in the first place. (Sometimes it might; in other cases, it may be a generic unit of reuse, such as a sorting routine or database interface.) We therefore believe the emphasis must shift from using terms like "modules" and "components" to discussing *what the modules encapsulate*.

Programs as Collections of Features

An end-user of a system typically does not care which database interface or sorting routine is used in the implementation, or even whether they are used at all; rather, users describe how they would like to see the system behave (via methods such as use cases) as a collection of units of functionality that we call *features*. When requirements change, they often either add or remove features, or change a previously identified feature. Managing changes to features is therefore a key problem in software development. If each feature is implemented by a specific module, it becomes easier to identify where changes must be made. Furthermore, if these modules meet the criteria of components laid out by Szyperski [5], then creating the system is just a matter of component composition, while adding and removing features is simply a matter of writing a new composition. In other words, this style of program organization would respect the feedback loop in the Jackson figure, observing that if the program's shape mirrors the requirements, changes in requirements will become easier to manage.

Writing these identifiable increments in conventional programming languages is challenging because an increment may affect parts of a program across traditional module boundaries: such increments are said to be *cross-cutting*. This observation has led to a growing body of work on developing new forms of program modularity [6–16] that support modularization around features and composition to create a variety of individual systems (thus forming a software *product line* [17]). Some techniques are purely static, effectively manipulating the program's source, while others have dynamic elements, offering the ability to reflect on the state of the program's execution and then to modify it. There is now a growing awareness of this style of programming (especially as popularized by "aspect-oriented programming"), and several case-studies highlight its feasibility and observe its benefits.

Research Program on Verifying Feature Modules

This model of program organization offers a substantial benefit to verification as well. One of the main challenges in modular verification is the decomposition of properties to align with the program module's boundaries. In theory, feature-orientation should largely eliminate this hurdle. As the user's perspective frames both the features (modules) and the properties, the scope of each property largely matches the scope of some module (with the exception of global system properties). In return, feature-orientation demands new theories of modular reasoning to support feature-based decompositions.

We have been working on techniques for modular model checking of feature-based designs since 2001 [18–20]. At a high level, this work shares the goals of other modular verification research, namely, to verify code fragments independently and derive some properties of the composed program from the properties of the fragments. The nature of feature-based design, however, adds some nuances to this problem.

1. Since features are added to programs to provide some user-defined functionality, we may need to prove that adding a feature (a) preserves established properties of programs, or (b) establishes a *new* property of the composed program. In some models, a feature might (c) establish a property that the original program should have satisfied, but did not.
   Performing these analyses modularly is important because we could potentially add many features to an existing program, at which point the cost of analyzing each possible product (a subset of features) becomes prohibitive. Feature-based design shifts the motivation for modular reasoning from making verification tractable in terms of computational resources to making it practical across the combinatorial number of products that can be built from designer-specified program modules.
2. Features can interact, causing properties of individual components to be violated in the composed program. In a telecommunications system, for example, a voice mail feature might be required to pick up an unanswered call after 4 rings and a call forwarding feature might be required to pass an unanswered call along after 4 rings. A system with both voice mail and call forwarding will respect only one of these requirements. Other examples are far more subtle. This *feature-interaction* problem is pervasive and not always amenable to formal analysis [21]. As model checking each combination of features for interactions is infeasible, modular analysis must support detecting those interactions that can be captured formally.

To date, our work has focused on property preservation (nuance 1a) with preliminary attention to feature interaction (nuance 2). We are able to model check CTL properties against individual features and perform lighter-weight checks to confirm that a feature's properties will be preserved when composed into an existing program. We are also able to modularly detect feature interactions that manifest as violations of properties expressed in CTL. Our work has identified several ways in which feature-based designs challenge the conventional assumptions of modular model checking:

1. Most modular verification work assumes parallel composition, while feature composition is largely (though not entirely) sequential. This in turn has interesting consequences:
   - Module composition can create new paths through programs (parallel composition deletes but does not add paths).
   - CTL appears more appropriate for modular reasoning than LTL (whereas LTL is arguably more effective under parallel composition [22]).
2. Modules are not closed because data from one module may persist into another. For instance, in an email system, one feature may encrypt a message and that attribute should persist into subsequent features, even though the models of those features do not mention encryption.
3. One module may refine the interpretations of propositions in another. E.g.: An email system may classify a message as anonymous if it has passed through an anonymous remailer, but adding a digital signing feature forces

the interpretation of anonymity to also require the message to be unsigned. Reinterpreted or persistent propositions often preclude lifting properties proven of an individual module to the composed program.

Even more fundamental, however, is our growing understanding that *verification may be the wrong problem to solve*. Traditional verification methods (especially model checking) are primarily designed to authoritatively determine the truth or falsity of properties over models. However, most of the property violations we observe arise only upon composition, because some compositions satisfy the properties while others fail them. Most verification runs over individual features are invariably inconclusive because there isn't enough information in the module to entirely satisfy or fail the property. This pushes the verification decision onto the composition step, which needs appropriate information that it can then use to perform a lightweight check that (a) is not too expensive, and (b) does not involve re-examining the innards of the individual modules. The appropriate analysis on individual modules is therefore some form of constraint generation, rather than outright verification. In our work on this approach [18] the constraints have been propositional and temporal, reflecting their foundation in model checking.

The need for constraint generation in practice does not contradict our earlier claim that properties align naturally with features. The truth of a property in a feature often depends on two specific pieces of information from the feature's environment: (1) whether control paths exiting the feature reach particular states in the original program, and (2) the values of persistent and reinterpreted data propositions determined in earlier features. Our proposed constraint generation is akin to generating environmental constraints; this step is feasible in our work because we generate environments specific to the program properties that a feature must preserve. The checks required to discharge these constraints at feature-composition time tend to be lightweight (simple propositional or reachability checks) because the feature that aligns with a property discharges the bulk of that property's obligations.

Some Challenges

Our observations leave open a large collection of interesting research problems. Some questions that need to be addressed include:

- Theoretical foundations for richer notions of composition. Most verification research is built on purely sequential composition or variations (synchronous, interleaved, etc) on parallel composition. The composition models in many feature-oriented programs lie between these extremes. We have studied programs that employ what we call *quasi-sequential composition* [19]: at the highest level, modules compose sequentially, but each module is formed of components that compose in parallel. This form of composition is interesting because it leads to states in the global state space that span different modules, but in controlled and predictable ways.

– Techniques for generating temporal constraints rich enough to support modular feature verification.
– Techniques for determining whether a code fragment introduces a desirable though previously untrue property over a program.
– Theories of compositional reasoning that are tuned for predicting feature interactions errors, as opposed to checking for success or failure of known properties.

Perspective

We find it telling that several researchers, working independently and in entirely distinct areas (often without any knowledge of each other), have within a few years proposed extremely similar models of software development centered around features. We believe that Jackson's picture explains why this model is not accidental, but rather fundamental to the way programs originate and evolve. Without lapsing into thoughts of silver bullets, we should take this model seriously, especially as formal programming language research is beginning to catch up with these less formal approaches (several papers [23–26] offer a representative sampling). Any attempt to lay the foundations for a practical program verifier must look ahead to how programs will be developed in the future, not only at programs written using antediluvian methods in legacy languages.

References

1. Jackson, M.: Why software writing is difficult and will remain so. Information Processing Letters **88** (2003) 13–25
2. Eliot, T.S.: The love song of J. Alfred Prufrock. In: Prufrock and Other Observations. The Egoist, Ltd., London (1917)
3. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems **17** (1995) 507–534
4. Xie, F., Browne, J.C.: Verified systems by composition from verified components. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, ACM Press (2003) 277–286
5. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (1998)
6. Aßmann, U.: Invasive Software Composition. Springer-Verlag (2003)
7. Batory, D.: Feature-oriented programming and the AHEAD tool suite. In: International Conference on Software Engineering. (2004)
8. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology **1** (1992) 355–398
9. Findler, R.B., Flatt, M.: Modular object-oriented programming with units and mixins. In: ACM SIGPLAN International Conference on Functional Programming. (1998) 94–104
10. Harrison, W., Ossher, H.: Subject-oriented programming: a critique of pure objects. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications. (1993) 411–428

11. Jackson, M., Zave, P.: Distributed feature composition: A virtual architecture for telecommunications services. IEEE Transactions on Software Engineering **24** (1998) 831–847
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming. (1997)
13. Lieberherr, K.J.: Adaptive Object-Oriented Programming. PWS Publishing, Boston, MA, USA (1996)
14. Mezini, M., Lieberherr, K.: Adaptive plug-and-play components for evolutionary software development. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications. (1998) 97–116
15. Smaragdakis, Y., Batory, D.: Implementing layered designs and mixin layers. In: European Conference on Object-Oriented Programming. (1998) 550–570
16. van Ommering, R.: Building Product Populations with Software Components. PhD thesis, Rijksuniversitat Groningen (2004)
17. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
18. Blundell, C., Fisler, K., Krishnamurthi, S., Hentenryck, P.V.: Parameterized interfaces for open system verification of product lines. In: IEEE International Conference on Automated Software Engineering. (2004)
19. Fisler, K., Krishnamurthi, S.: Modular verification of collaboration-based software designs. In: Symposium on the Foundations of Software Engineering, ACM Press (2001) 152–163
20. Li, H.C., Krishnamurthi, S., Fisler, K.: Modular verification of open features through three-valued model checking. Automated Software Engineering **12** (2005) 349–382
21. Keck, D.O., Kuehn, P.J.: The feature and service interaction problem in telecommunications systems: A survey. IEEE Transactions on Software Engineering **24** (1998) 779–796
22. Vardi, M.Y.: Branching vs. linear time: Final showdown. Available at `http://www.cs.rice.edu/ vardi/papers/index.html` (2001)
23. Ancona, D., Lagorio, G., Zucca, E.: Jam—designing a Java extension with mixins. ACM Transactions on Programming Languages and Systems **25** (2003) 641–712
24. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1998) 171–183
25. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004)
26. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: European Conference on Object-Oriented Programming. (2003) 248–274