Transformation Systems at NASA Ames

Wray Buntine	Univ of Calif. Berkeley	wray@eecs.berkeley.edu
Bernd Fischer	NASA Ames/RIACS	fisch@ptolemy.arc.nasa.gov
Klaus Havelund	NASA Ames/RECOM	havelund@ptolemy.arc.nasa.gov
Michael Lowry	NASA Ames	lowry@ptolemy.arc.nasa.gov
Tom Pressburger	NASA Ames/RECOM	ttp@ptolemy.arc.nasa.gov
Steve Roach	Adams State College	smroach@adams.edu
Peter Robinson	NASA Ames/RECOM	robinson@ptolemy.arc.nasa.gov
Jeffrey Van Baalen	Univ. Wyoming	jvb@uwyo.edu

ABSTRACT

In this paper, we describe the experiences of the Automated Software Engineering Group at the NASA Ames Research Center in the development and application of three different transformation systems. The systems span the entire technology range, from deductive synthesis, to logic-based transformation, to almost compiler-like source-to-source transformation. These systems also span a range of NASA applications, including solving solar system geometry problems, generating data analysis software, and analyzing multithreaded Java code.

Keywords

deductive synthesis, source-to-source transformation, rewriting, compilation

1 INTRODUCTION

In this paper, we describe the experiences of the Automated Software Engineering Group at the NASA Ames Research Center in the development and application of three different transformation systems. The systems span the entire technology range from deductive synthesis, to logic-based transformation, to almost compiler-like source-to-source transformation.

Deductive synthesis systems [11] have the same goal as "classical" (i.e., non-deductive) transformation systems, namely the (mostly automatic) translation of non-executable specifications into executable programs. They use, however, a different technology. The specification is recast as a theorem in the first-order predicate calculus and submitted to an automatic theorem prover. The resulting program is then extracted from the proof which must thus be constructive.

Deductive synthesis is not an alternative to but an instance of the general transformational software development paradigm. This is a consequence of the dual nature of proofs. On the one hand, a proof, especially a tableau-style proof [12], is a stepwise transformation of a theorem into axioms. On the other hand, it can be considered as a program, due to the Curry-Howard isomorphism [7].

The main difference between the classical transformation and deductive synthesis approaches is the question of *control*, i.e., who is in charge to control the transformation/proof process? In a classical transformation system, the system designer is in charge. Usually, the transformation process is constrained by organizing the set of transforms into different, fixed, mutually distinct layers; these layers are executed in a fixed order and transforms in later layers are generally more detailed and more localized. In a purely deductive system, the theorem prover is in charge. The proof process is only restricted by properties of the applied calculus; essentially, every transform can be applied at every possible locale, at any time. Hence, a transformation system may in turn also be considered to be an *operationalization* of a deductive synthesis system.

A syntax-directed source-to-source transformation system (in other words, a compiler) can be considered as the ultimate operationalization of a transformation system: the search for applicable transformations (i.e., transforms and locales) is completely eliminated, either by differentiation of the syntax which renders the choice of locales irrelevant, or by completion of the set of transforms which renders the choice of transforms irrelevant.

The application of transformational techniques at NASA is driven by the need to achieve goals in software development: (*i*) lowering the development cost, (*ii*) lowering the required expertise, and (*iii*) validating and verifying the developed system. These goals, however, also apply to the development of transformation systems.

The only variable cost in the development of a transformation system is the operationalization step. Since deductive synthesis systems need in principle no explicit operationalization, they are prime candidates. However, non-operationalized deductive systems, hit the complexity barrier inherent to theorem proving before they scale-up to non-toy problems. We have thus developed automated techniques which facilitate a deductive operationalization and at the same time keep the level of required expertise within reasonable bounds.

2 AMPHION

Amphion [9] implements synthesis-based reuse of library subroutines. It translates a problem specification expressed in logic into a Fortran program that solves the problem. The Fortran program at present is a straight-line program that makes calls to JPL's NAIF library subroutines, which are in the domain of solar system geometry. Amphion/NAIF deals with problem specifications of about 30 lines, generating about 100 lines of Fortran code. In a smaller experiment, Amphion has been applied to airplane gridding subroutines used in Computational Fluid Dynamics. Amphion's translation process comprises three main phases.

1. Rewrite the abstract specification containing abstract operations to a canonical set of abstract operations (e.g. angle-between-two-planes to angle-between-the-two-normal-vectors)

2. Use deductive synthesis to solve the abstract specification; this yields a term over concrete operators.

3. Translate the concrete term to a level isomorphic to Fortran, whence it is printed out.

The first two phases are performed together by SNARK, a resolution theorem prover developed at SRI by Mark Stickel [13].

The theorems used in deductive synthesis in the second phase above capture three kinds of facts.

$$R_a(x, F_a(x)) \quad (1)$$

$$F_a(A_1(c)) = A_2(F_c(c)) \quad (2)$$

$$A_1(c) = A_2(\mathrm{convert}_{1,2}(c)) \quad (3)$$

The first expresses solutions to constraints (1); e.g., that $F_a(c)$ is a solution for y to $R_a(c, y)$. The second are homomorphisms that implement an abstract operator F_a as a concrete operator F_c (2). The third are conversion axioms that convert a concrete value from one representation of an abstract value into another representation (3).

In SNARK, control of the deductive process is effected by specifying a search strategy (we chose set-of-support), Recursive Path Orderings (RPO), and a function that orders the goal clauses on the agenda. The RPO specifies whether an equation can be used as a rewrite rule, and whether a paramodulation (an equality inference rule used in resolution theorem provers) should occur. However, Amphion loses completeness because it uses RPOs together with the set-of-support strategy. This means that it might, in theory, fail to synthesize a program for a given specification; in practice, however, this has not been a problem.

The deductive process generally proceeds in two steps: initially, the constraints are solved abstractly in terms of abstract operators using the first sort of fact; then concrete operators are found that implement the abstract operators using the second sort of facts oriented left-to-right and the third sort of facts acting as glue between the second sort of facts.

Concocting an agenda ordering function that ordered the tasks as described above was difficult, requiring reasoning about the theorem proving process and the axioms. Meta-Amphion [10, 15] was invented to deal automatically with this problem. It finds subtheories that can be replaced by decision (satisfiability) procedures; it can be thought of as a theory compiler. The resulting system can replace several non-deterministic inference steps by a single deterministic computation step. This puts less reliance on the agenda ordering function.

This makes it feasible for a domain expert (i.e., someone familiar with the subroutine library but not automated deduction) to write such axioms as (1,2,3). It is a design goal of Amphion that the people developing the library can extend Amphion to make the library more accessible to customers. The reason Amphion is successful is that, due to the use of deductive synthesis, each axiom can be reasoned about in isolation. A user interface enabling a domain expert to enter axioms is being built now. An RPO that orients an equation like (2) left-to-right can be generated automatically.

For the automated reasoning about the correctness of axioms, a type checker was written that applied to the axioms and rewrite rules. We also experimented with running a completion procedure, but not thoroughly: we got lost trying to orient the equalities with which it came up.

Translation Phase

The translation phase applies several passes of correctness-preserving transformations to the answer term returned by SNARK. The following tasks are accomplished: variables are introduced for subterms (this is required to pass data from one Fortran CALL statement to the next); parameters and dummy parameters are correctly assigned to subroutines that return multiple values; and parameter passing conventions used by NAIF are implemented. The reason for the multi-pass architecture is separation of concerns. Because it is written as a transformation system, rather than a monolithic piece of code, we hope that it will be able to handle extensions, such as conditional statements;

however, this has not been tested.

SNARK uses special-purpose formula data structures; the latter transformation phases use LISP lists. The latter phases were first written using Software Refinery, but were later transliterated into LISP. The Refine transformations became either LISP functions if they were complicated, e.g. rule schemas, or rewrite rules if they were simple enough. If a transformation is complicated, it is not analyzable, e.g. by the type checker. The following are instances of two rule schemas; the general schemas cannot be expressed as simple rewrite rules.

```
(get-field 2 (tuple x y z)) --> y

(let ((x E1) ...) bdy)
  -->
(letp ((x ...) suchthat (and (= x E1) ... )) bdy)
```

Tracing

Tracing information was added to SNARK and the translation process to record where each rewrite and inference rule applied [16]. This is used to add comments to code as to which abstract variable a concrete variable corresponds.

Abstractly, an explanation is a set of connections between the Fortran program and the problem specification or parts of the domain theory. The explanation generation process starts with the Fortran abstract syntax term, considered to be the root of a derivation tree, and traces backwards through the derivation to the specification and domain theory, considered to be the leaves of the derivation tree. Both the proof, resulting in an applicative answer term, and the trace of the program transformations, resulting in a Fortran abstract syntax tree, are part of this derivation tree.

More formally, a derivation is a directed acyclic graph whose nodes are derivation and transformation steps and whose arcs encode the "derived from" relation. Each derivation step is a triple $\langle F, T, A\rangle$, where A is an inference rule application, and F and T are the resulting formula and answer term, respectively. Each application of a rule specifies: the inference rule that was applied (one of transformation, demodulation, paramodulation, or resolution); the input formulae $F_1, \ldots, F_n$; and the locations (path in the formulae) of the subterms to which the rule was applied.

An explanation is an equality connecting locations in the Fortran abstract syntax term to constructs in the specification and domain theory, from which it was derived. Such an equality is computed in two steps. First, explanation equalities of each step of the derivation are extracted. Explanation equalities are equalities between locations in a formula F and locations in the parent formulae $F_1, \ldots, F_n$ in the derivation. Second, an equality is computed between the relevant location in the Fortran abstract syntax term and locations in the leaves of the derivation.

3 PROBABILISTIC NETWORKS

The motivation for this work was to enable the quick development of small and efficient data analyzers that could be run on a Martian rover that uses a spectrometer to classify minerals. The relevant type of data analysis here is parameter estimation, where, given data that is drawn from a statistical model (i.e., a parameterized distribution), the task is to find the parameter values that maximize some measure of fitness of the data to the model.

For such applications, experienced statisticians iteratively experiment with different statistical models of the domain to be analyzed and with different algorithms to be employed to do the analysis. However, the implementaion of—even a prototype—data analysis program for each model is too expensive, which hinders the experimentation with different models and algorithms. Moreover, turning these prototypes into realistic, space- and time-efficient code introduces another bottleneck. Thus, a program transformation system which translates a statistical model into an efficient program (as suggested in [1]), would be valuable tool for the statistical data analysis domain.

We are currently developing the system PN (Probabilistic Networks) [2] which is intended to be such a tool. It uses a specification language which is based on Bayesian networks. This framework provides a way of deriving the full joint probability over variables, and various conditional and marginal probabilities as required to synthesize the analysis programs. Bayesian networks are a standard notation in the data analysis domain and have also been adopted for other successful data analysis systems such as BUGS [14]. In the PN specification language the variables involved in the problem are defined together with their properties and inter-dependencies. For example, a variable can be declared as a constant, to depend deterministically on other variables, or to be distributed according to a distribution parameterized by other variables.

The PN system generates pseudocode as an intermediate stage before translation to Java. The pseudocode may embody efficient algorithm templates (e.g., clustering algorithms) but may also call routines for doing numerical optimization.

The program synthesis process is triggered by a parameter estimation problem statement which asks to find the values of some of the variables that optimize a given probability given data for other variables. The synthesis process itself transforms the network and the problem statement. Subproblems are generated based on theorems about how the network can be decomposed.

Some of the problems can be solved in closed form, some by calling numerical optimization or solution subroutines, and some by using general purpose statistical algorithms; the transforms associated with these problems return code fragments. These subproblems may also involve probability expressions which can be simplified based on further domain-specific theorems. Some other subproblems require further, recursive decomposition of the network; the associated transforms return simpler (network, problem statement)-pairs.

All transformations are expressed as theorems in Horn-clause logics; this facilitates reasoning about their correctness. For efficiency reasons, we use a Prolog-interpreter as a deductive execution machine (i.e., metaprogram); each transform becomes a rule which checks the preconditions of the underlying theorem and builds up pseudocode as it decomposes the network and problem specification.

The final phase, which we have not finished, is a transformation from pseudocode to Java. Here, issues such as numerical stability and optimization need to be addressed.

4 JAVA PATHFINDER

One of our goals is to apply and develop verification tools based on model checkers, theorem provers, and static analysis in general. A verification tool typically analyses a term in some language (a specification, a design or a concrete program) with the purpose of deciding whether the term satisfies some given properties. This provides a complementary and higher level of debugging and V&V than traditional testing methods. The benefit becomes particularly apparent when dealing with non-deterministic systems, for example concurrent and distributed programs, where testing techniques typically cannot control the scheduling of interleaved processes. That is, using traditional testing, the successful run of a particular test suite will not be convincing due to the fact that the test run may have missed certain possible interleavings of the involved processes. A verification tool, such as a model checker, will typically explore all possible interleavings.

Almost all existing verification tools, however, analyze terms in some special purpose language particular for that tool. We believe that verification tools will be accepted and used by NASA programmers only if they can analyze programs/designs/specifications written in the languages already used by these programmers. Therefore we have started a general effort to bridge the gap between frequently used languages and these special purpose verification languages in order to benefit from the analytic engines underneath the latter. In particular, we are developing a translator called Java PathFinder (JPF) [4] from the Java programming language to the Promela language of the Spin model checker.

Spin [6] is a verification system that supports the design and verification of finite state asynchronous process systems. Programs are formulated in Promela, which is a simple multi-threaded programming language with non-deterministic guarded commands. Processes communicate either via shared variables or via message passing through buffered channels. Properties to be verified are stated as assertions in the program or as formulae in the linear temporal logic LTL. The Spin *model checker* can automatically determine whether a program satisfies a property, and in case the property does not hold, it generates an error trace.

One of our efforts to formally verify a multi-threaded operating system for the DEEP SPACE 1 spacecraft is documented in [3]. The operating system is implemented in a multi-threaded version of Common LISP. The verification effort consisted of hand-translating parts of the LISP code into the Promela language of Spin. A total of 5 errors were identified, a very successful result. It was, however, clear that hand-translating such amounts of code is impractical, and this motivated the translator described here.

The Source Language

The intended source language of the translation is Java as a general purpose programming language, rather than as a specialized World Wide Web applet programming language. A significant subset of Java version 1.0 is currently supported by JPF: dynamic creation of objects with methods and data such as integers, booleans, arrays and object references. Furthermore class inheritance, dynamically created threads and synchronization primitives for modeling monitors (`synchronized` statements, and the `wait` and `notify` methods), exceptions, thread interrupts, and most of the standard programming language constructs such as assignment statements, conditional statements and loops. However, the translator is still a prototype and misses some features, such as packages, overloading, method overriding, recursion, strings, floating point numbers, static variables and static methods. Finally, we do not translate the predefined class library. A well-formedness predicate is applied to the Java program before the translation to determine whether it falls in the subset of translatable programs.

A Java program to be translated will contain temporal logic specifications stated as calls to special static methods (like "`Verify.assert(alarm == false)`") defined in a predefined class called `Verify`. When translating the Java program these calls will be translated into the assertion language of Promela or into LTL formulae. By defining the special `Verify` class and defining temporal logic operators as Java methods (with possibly empty bodies since they mainly have importance for the translation) we avoid extending the Java language with specification constructs.

The Translator
The translator is written using Moscow ML (for parsing) and Common LISP for the translation. The result of the parsing is an abstract syntax tree represented as a LISP S-expression. A set of LISP macros using the object oriented features of LISP have been defined to allow pattern matching over this tree, since LISP does not have pattern matching.

The translator is defined by recursive descent and pattern matching over the tree, printing Promela code directly to an output file. Hence, the translation does not generate intermediate code – the translation is direct, source to target in one go. The generated code is, however, divided into C macro definitions and the Promela code itself (Promela code is allowed to contain calls of such macros, which will be expanded out). The macros typically model Java kernel operations such as applying synchronization locks on objects, throwing exceptions and handling object references. The generated Promela code can therefore be regarded as belonging to an enriched Promela, augmented with these kernel operations, and the following macro expansion will then result in lower level Promela code.

The translation functions all have access to the complete abstract syntax tree in addition to the portion of the tree they are respectively designed to transform. This is in contrast to maintaining a symbol table that grows as the tree is descended. Just keeping the tree around seemed simpler than maintaining a nearly isomorphic symbol table.

The translator does not keep a record of the applied transformations (LISP functions). However, traceability between the source Java code and the target Promela code is essential when the Spin model checker generates an error trace caused by a broken temporal property. This error trace can be simulated in Spin, illustrating the steps that lead to the error. However, a simulation is needed at the Java source code level, and this requires a mapping from Spin error traces to corresponding Java error traces. We don't have this mapping implemented for the moment. Instead, the Java program is allowed to contain calls of `Verify.print(...)` statements that will cause printing on graphical message sequence charts when running the error trace in Spin.

Correctness and Scalability
The translator itself is a prototype experimenting with the idea of transforming a real programming language into a model checker language. Since this is a novel and challenging problem in itself we have not spent efforts to prove the translation correct. Our main goal is to see whether this approach can have practical importance. The translator is expected to be scalable to almost full Java 1.0.

A major challenge, however, is whether model checking of the resulting *target* Promela program scales up to large programs. A 1000 line Java program has been analyzed using the tool, confirming the existence of a deadlock [5]. In order to handle substantially larger programs one needs to apply transformations to the Java source program in order to remove details that are not important for the properties to be verified. Together with Stanford University and Kansas State University we are currently exploring techniques such as program abstraction, program slicing, and static analysis in general. Some of these techniques can be regarded as program transformation schemes that produce target programs with a smaller state space than the source programs. The intention is to apply such transformations on a Java program before the translation into Promela is applied. In a current project, a 13 K line satellite file communication protocol written in Java is being analyzed. This work will identify useful abstraction techniques.

An alternative solution to deal with scalability is compositional model checking, where only smaller portions of code are verified at a time, and where the results are then composed to deduce the correctness of larger portions of code. Imagine for example that one has programmed a Java class modeling a bounded buffer which is to be used in a multi threaded context. We can then analyze its correctness by "putting it in parallel" with an aggressive environment consisting of producer and consumer threads, and we can then analyze the properties of this small system. At least one of the errors in the Deep Space 1 operating system could have been found this way.

A different issue is the treatment of dynamic object creation. Currently the heap is modeled as a fixed size array. We are studying how to model a dynamic heap and garbage collection in a model checker. Note that no alias analysis is done at this point. It will, however, play a role in optimizing the verification.

5 CONCLUSIONS
In this paper we have summarized a variety of transformation systems developed at NASA Ames for the purpose of automating aspects of software engineering. Our transformation systems lower the cost of developing software, lower the cost for maintaining software, lower the expertise required to develop software, and provide more effective ways to verify and validate software systems.

To achieve these goals, we employ a spectrum of transformational technology, ranging from transformation systems organized as a compiler to deductive synthesis. The choice of transformation technology depends upon the characteristics of the source to target translation problem. When the translation problem can be decomposed along the lines of syntactic constructs, then syntax-directed source-to-source transformation systems (non-optimizing compilers) are highly efficient and relatively easy to control. For the first generation of the Java PathFinder, this technology worked well; enabling rapid experimentation of different ways to embed Java syntactic constructs into Promela. When the translation problem is characterized by a sequence of intermediate forms, then a sequence of sets of transformations (applied to exhaustion at each stage) works well. As long as the number of transformations applied at each stage is small, then issues such as termination - i.e., a well defined ordering ensuring progress towards the next intermediate form - are manageable, but require more expertise to control than compiler technology. The interaction of the transformation rules requires care to ensure correctness. Deductive synthesis technology offers the potential to develop transformation systems not by defining 'how' constructs are transformed from source to target, but rather the declarative relationship between source and target. For small domains, the deduction engine will find the right path for transforming from source to target by exploring the possibilities denoted by the declarative relationship. However, as the domain becomes larger, this exploration becomes combinatorially prohibitive. This leads to the goal of operationalizing a deductive synthesis system.

Despite the effectiveness of these systems, we perceive barriers to the more wide-spread use of transformational technology. To break through these barriers, the same goals that apply to improving software development in general also need to be achieved to improving the development process for transformation systems: lower the cost of developing a transformation system, lower the cost to modify and extend a transformation system, lower the expertise required to develop a transformation system, and provide effective ways to verify and validate transformation systems.

ACKNOWLEDGEMENTS

We would like to acknowledge the efforts of Phil Oh who helped create a Java-based graphical interface to Amphion, and Santos Lazzeri who created a text interface to Amphion [8], and a graphical interface to Meta-Amphion.

REFERENCES

[1] W. Buntine. Operations for learning with graphical models. *J. AI Research*, 2:159–225, 1994.

[2] W. Buntine, B. Fischer, and T. Pressburger. Towards automated synthesis of data analysis programs. *Submitted to IJCAI–99*, 1999.

[3] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proc. 4th SPIN workshop*, 1998.

[4] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. To appear in *Intl. J. Software Tools for Technology Transfer*, February 1999.

[5] K. Havelund and J. Skakkebaek. Practical Application of Model Checking in Software Verification, 1999. Submitted for publication.

[6] G. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[7] W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pp. 479–490. Academic Press, 1980.

[8] S. Lazzeri. A comparison of different HCI styles in a KBSE system. In *10th Intl. IFIP WG 5.2/5.3 Conf. PROLAMAT 98*, Trento, Italy, September 1998.

[9] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Proc. 9th Knowledge-Based Software Engineering Conf.*, pp. 48–57, 1994.

[10] M. Lowry and J. Van Baalen. Meta-Amphion: Synthesis of efficient domain-specific program synthesis systems. *J. Automated Software Engineering*, 4:199–241, 1997.

[11] Z. Manna and R. J. Waldinger. Fundamentals of Deductive Program Synthesis. *IEEE Trans. Software Engineering*, SE-18(8):674–704, 1992.

[12] R. M. Smullyan. *First-Order Logic*. Springer, 1968.

[13] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, (ed.), *Proc. 12th Intl. Conf. Automated Deduction, LNAI 814*, pp. 341–355. Springer, 1994.

[14] A. Thomas, D. J. Spiegelhalter, and W. R. Gilks. BUGS: A program to perform Bayesian inference using Gibbs sampling. In *Bayesian Statistics 4*, pp. 837–842. Oxford University Press, 1992.

[15] J. Van Baalen and S. Roach. Using decision procedures to build domain-specific deductive synthesis systems. In *LOPSTR'98: Proc. of the 8th Intl. Workshop on Logic Program Synthesis and Transformation*, 1998.

[16] J. Van Baalen, P. Robinson, M. Lowry, and T. Pressburger. Explaining synthesized software. In D. F. Redmiles and B. Nuseibeh, (eds.), *Proc. 13th Intl. Conf. Automated Software Engineering*, pp. 240–248, 1998.