Practical and Verifiable C++ Dynamic Cast for Hard Real-Time Systems

Damian Dechev, Rabi Mahapatra, and Bjarne Stroustrup
Texas A&M University, College Station, Texas 77843, USA
dechev@tamu.edu; {rabi, bs}@cs.tamu.edu

Received 2 July 2008; Accepted 16 September 2008

The dynamic cast operation allows flexibility in the design and use of data management facilities in object-oriented programs. Dynamic cast has an important role in the implementation of the Data Management Services (DMS) of the Mission Data System Project (MDS), the Jet Propulsion Laboratory's experimental work for providing a state-based and goal-oriented unified architecture for testing and development of mission software. DMS is responsible for the storage and transport of control and scientific data in a remote autonomous spacecraft. Like similar operators in other languages, the C++ dynamic cast operator does not provide the timing guarantees needed for hard real-time embedded systems. In a recent study, Gibbs and Stroustrup (G&S) devised a dynamic cast implementation strategy that guarantees fast constant-time performance. This paper presents the definition and application of a co-simulation framework to formally verify and evaluate the G&S fast dynamic casting scheme and its applicability in the Mission Data System DMS application. We describe the systematic process of model-based simulation and analysis that has led to performance improvement of the G&S algorithm's heuristics by about a factor of 2. In this work we introduce and apply a library for extracting semantic information from C++ source code that helps us deliver a practical and verifiable implementation of the fast dynamic casting algorithm.

Categories and Subject Descriptors: Programming Tools and Techniques [**Programming Language**]

General Terms: C++ Programming Techniques, Embedded Flight Software, Program Verification

Additional Key Words and Phrases: Constant Time Dynamic Cast, Autonomous Embedded Systems, Model-based Software Development, Static Analysis

1. INTRODUCTION

ISO Standard C++ [ISO/IEC 14882 International Standard 1998] has become a common choice for hard real-time embedded systems such as the Jet Propulsion

Laboratory's Mission Data System [Ingham et al. 2004]. This is so because ISO C++ offers efficient abstraction model, good hardware use, and predictability. C++'s model of computation has helped engineers deliver more correct, maintainable, and comprehensible software compared to code relying on lower-level programming concepts [Stroustrup 2004]. However, several C++ features are usually considered unsuitable for programming real-time systems because they do not guarantee predicable constant-time performance [Goldthwaite 2006]. ISO C++ does not provide the necessary timing guarantees for free store (heap) allocation, exception handling, and dynamic casting. In particular, the most common compiler implementations of the dynamic cast operator traverse the representation of the inheritance tree (at run time) searching for a match. Such implementations of dynamic cast are not predictable and are unsuitable for real-time programming. Gibbs and Stroustrup (G&S) [Gibbs and Stroustrup 2006] describe a technique for implementing dynamic cast that delivers significantly improved and constant-time performance. The key idea is to replace the runtime search through the class hierarchy with a simple (constant-time) calculation, much as the common implementations of the C++ virtual function calls search the class hierarchy at compile time to reduce the runtime action to a simple array subscripting operation. In the G&S scheme, a heuristic algorithm assigns an integer type ID at link time to each class. The type ID assignment rules guarantee that at run time a simple modulo operation can reveal the type information and check the validity of the cast. The requirements for the heuristics assigning the type IDs are that:

(1) They must keep the size of the type ID to a small number of bits.A 64-bit type ID should be sufficient for very large class hierarchies
(2) Avoid conflicts and type ID assignments that create ambiguous or erroneous type resolution at run time
(3) Handle virtual inheritance

There are four heuristic schemes and a few possible optimizations suggested in [Gibbs and Stroustrup 2006]. However, none of those heuristics guarantee the best solution for every possible class hierarchy. The quality of the type ID assignment heuristics has a critical importance for the performance of the G&S scheme. With better heuristics, a smaller type ID size would be sufficient to facilitate complex and large class hierarchies that would certainly need a significantly bigger type ID size with a poor assignment scheme. The main contribution of this work is to present how the algorithm optimization problem encountered has been successfully automated and moreover that its automation has led us to quick but significant improvements of the initial scheme. To guarantee a practical and verifiable implementation of the fast dynamic casting scheme, we introduce and apply our innovative expression template [Veldhuizen 1995]-based approach for extracting semantic information from the C++ source code.

As pointed out by Lowry [Lowry 2002], the increasing complexity of future space missions, such as the Mars Science Laboratory [Volpe 2005] and Project Constellation [Stoica et al. 2005], raises concerns whether it is possible to establish their reliability in a cost-effective manner. Lowry's analysis indicates that at the present moment the

verification and certification cost of mission critical software exceeds its development cost. Perrow [Perrow 1999] studies the risk factors in the modern high technology systems. His work identifies two significant hazard dimensions: *interactions* and *coupling*. Complex interactions represent unexpected and unknown sequences and thus cannot be entirely comprehensible at the time of system development. A tightly-coupled system has a number of time-dependent processes that cannot tolerate delays. Perrow classifies space missions in the riskiest category since both hazard factors are present. The dominant paradigms for software development, assurance, and management at NASA rely on the principle "estwhat-you-fly and fly-what-you-test". Born out of experience and hindsight, this methodology had been applied in a large number of robotic space missions at the Jet Propulsion Laboratory. For such missions, it has proven suitable in achieving adherence to some of the most stringent standards of man-rated certification such as the DO-178B [RTCA 1992], the Federal Aviation Administration (FAA) software standard. Its Level A requirements demand 100% coverage of all high and low level assurance policies. However, the present certification methodologies are prohibitively expensive for systems of high complexity [Schumann and Visser 2006].

In this paper we present a co-simulation framework based on the SPIN model checker [Holzmann 2003] to simulate, evaluate, and formally verify the G&S fast dynamic casting algorithm and its application in mission critical code such as the Data Management Services [Wagner 2005] of the Mission Data System. The aim of the Mission Data System is to provide a unified state-based and goal-oriented architecture for building complete data and control systems for autonomous space missions. The framework's state-and model-based methodology and its associated systems engineering processes and development tools have been successfully applied on a number of test systems including the physical rovers Rocky 7 and Rocky 8 and a simulated Entry, Descent, and Landing (EDL) system for the Mars Science Laboratory mission. We use the feedback from the model checker to perform systematic analysis of the G&S scheme and look for improvements to the heuristics for type ID assignment. SPIN is an on-the-fly, linear-time logic model-checking tool that was designed for the formal verification of dynamic systems with asynchronously executed processes. Model-checking tools have been widely applied for the verification of a large variety of systems, including flight software [Gluck and Holzmann 2002], network protocols [Musuvathi and Engler 2004], and scheduling algorithms [Ruys 2003]. We are unaware of work suggesting its use for the analysis and optimization of compiler heuristics. Compiler verification usually focuses on seeking a proof on the preservation of the program semantics during the various compiler passes [Lerner et al. 2003]. Our work presents the application of a model-checking tool for the analysis and refinement of the combinatorial optimization problem posed by the G&S type ID assignment scheme. Our co-simulation framework consists of the following components:

(1) An abstract model of the G&S type ID assignment heuristics
(2) A procedure for exhaustive search of the state space discovering the best type ID assignment

The analysis of the heuristics simulation performed in SPIN provides us with ideas of possible improvements to the G&S type ID assignment. We include and evaluate the proposed improvements in the abstract model in order to seek refinement of the G&S type ID assignment scheme. The experiments we have executed show that the G&S priority assignment is not optimal with respect to the best possible type ID assignment where non-virtual multiple inheritance is used. While potentially dangerous if not constructed carefully, such hierarchies happen to be of significant practical importance [Stroustrup 2000]. Based on our experiments, we suggest optimizations that lead to significant improvement of the G&S heuristics performance. We rely on model checking for the validation, simulation, and analysis of the fast dynamic casting algorithm. Due to the heavy computational overhead and the state space explosion problem, the application of formal verification techniques is limited to abstract models of the system's design. In this work we introduce and apply an innovative expression-template based technique for extracting semantic information from the C++ source code in order to deliver a practical and verifiable implementation of the G&S fast dynamic casting scheme. This paper makes the following contributions:

(1) Introduces the use of a co-simulation framework based on model-checking for the analysis and improvement of a compiler-heuristics optimization problem
(2) Verifies and analyzes the G&S C++ fast dynamic casting scheme and its application in mission critical code such as the MDS Data Management Services
(3) Implements optimizations to the G&S heuristics leading to the discovery of optimal type ID assignment in 85% of the class hierarchies, in contrast to 48% for the original G&S algorithm
(4) Presents the design and application of an innovative expression template -based approach for extracting semantic information from the application's source code in order to guarantee a practical and verifiable implementation of the fast dynamic casting scheme

The rest of the paper is organized as follows: section 2: a brief description of the G&S fast dynamic cast algorithm, section 3: our approach to co-simulation and improvements to the G&S heuristics, section 4: discussion on the challenges of mission critical code and the applicability of the G&S dynamic cast section 5: performance results for the G&S algorithm and the proposed improvements, section 6: design and implementation of Basic Query: a library for extracting semantic information from C++ programs and its application for delivering a practical and verifiable fast dynamic cast operation, and section 7: conclusion.

2. FAST DYNAMIC CASTING ALGORITHM

The G&S fast constant-time implementation of the dynamic cast operator works as follows: at link time, a static integer type ID number, preferably 32 or 64-bit long, is assigned to each class. The ID numbers are selected so that the operation id_a *modulus* id_b yields zero if and only if the class with id_a is derived from the class with id_b. This is done by exploiting the uniqueness of factorization of integers into prime factors. Each class is assigned a *key* prime number. The *type ID* of a class is calculated by

multiplying its *key number* with the key numbers of each of its base classes. In the cases where a class contains more than a single copy of a base class, the type ID is computed by taking the square of the corresponding base class ID. The only constraint of the approach is the desire to limit the ID size to fit the machine's built-in integer types. The key primes are not required to be unique and the same prime key can be used for classes that belong to different groups (i.e. do not share common descendants). Gibbs and Stroustrup suggest four approaches for assigning the type IDs in a space-efficient manner. Each method is based on a preliminary computation of the priority factor of each class. The priority reflects the class impact on the growth of the type ID numbers in the hierarchy. Thus, classes with greater number of descendants should receive higher priority and smaller key prime number values respectively. The four possible schemes suggest that:

1. The priority of a class is the *maximum* number of ancestors that any of its descendants has. This scheme was chosen for the initial implementation and testing of the G&S algorithm and also closely followed in the implementation of the abstract model used for our simulation
2, 3, 4. If a range of primes is assigned to every level with wider levels receiving larger initial values, then each node could be assigned an additional value that is proportional to the logarithm of the (*2. minimum, 3. mean, 4. maximum*) prime in its level. Priorities of hierarchy leaves are computed by taking the sum of these additional values for the leaf itself and all of its ancestor classes

After the priority of each class has been computed, the classes with the highest priority get the smallest prime numbers. According to this scheme, prime numbers can be reused only if there are two classes on the same level of the class hierarchy and only if they do not share common descendants, they are not siblings, and also that none of their parents share a common descendant. According to the ID assignment rules, we know that:

(1) $idx = kx \times (ka)^2 \times ka_1 \times (kb)^2 \times kb_1 \times kc$
(2) $idy = ky \times kc \times kc_1 \times (kd)^2 \times kd_1 \times kb$
(3) $idz = kz \times kd \times kd_1 \times kc$

As an example, let us consider the class hierarchy presented in Figure 1. Given a set $S_{classes}$ with 11 classes in the hierarchy and the set of the first 11 prime numbers $P = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31\}$, we must assign each class V a key $k_v \in P$ such that, the maximum of the set $id_{leaf} = \{idx, idy, idz\}$, the set consisting of the ID numbers of all leaf nodes in $S_{classes}$, is minimal. As we already know, prime numbers need not be unique for each class and can be reused in same circumstances.

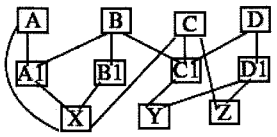


Figure 1. A class hierarchy with 11 classes.

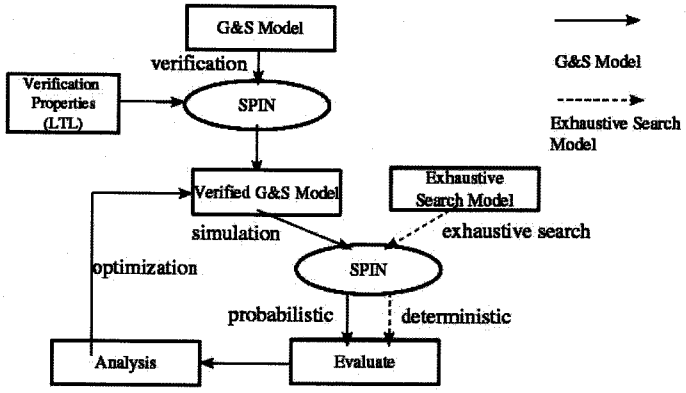


Figure 2. A co-simulation framework for G&S improvement and verification.

3. A CO-SIMULATION FRAMEWORK

The goals of the co-simulation framework are to validate the main invariants of the G&S heuristics, improve its performance, and establish its applicability in mission critical systems. The co-simulation process in the framework (Figure 2) consists of three consecutive stages: verification, evaluation, and analysis. The verification phase is a straightforward application of model checking where an abstract description of the system's behavior is checked against a set of invariants. In the evaluation stage the simulation results from the probabilistic approach are contrasted to the outcome of the deterministic approach. The aim of the analysis stage is to closely examine the instances where the solutions yielded by the two implementations differ. We identify

Algorithm 1 Pseudocode of the co-simulation approach.

```
1: const int MAX_NUMBER_TESTS
2: VERIFY :
3: repeat
4:    Formal Verification (G&S Model) → error report
5:    if (no errors) then
6:       goto EVALUATE
7:    else
8:       study counter example
9:       correct G&S
10: until TRUE
11: EVALUATE :
12: count = 0
13: for (count < MAX_NUMBER_TESTS) do
14:    Simulation(G&S Model) → G&S solution
15:    Enumeration(Exhaustive Search Model) → best solution
16:    if (G&S solution ≠ best solution) then
17:       add instance to SIS
18:    count + +
19: ANALYZE :
20: for all i ∈ SIS do
21:    look for a pattern
22:    modify G&S
23:    goto EVALUATE
```

patterns among the inconsistent results that reveal the weaknesses of the probabilistic solution. The framework works by executing two independent models, the G&S model and the exhaustive search model. The first input component to the co-simulation framework (Figure 2) is an abstract model of the G&S fast dynamic casting heuristics, implemented in Promela (SPIN's input language) and the embedded C primitives it allows. The G&S abstract model is subsequently used to verify the main invariants of the G&S heuristics and at the same time provide us with a simulation testbed to examine the heuristics performance. The second component of the framework is the exhaustive search model that simply looks into all possible type ID assignments to discover the optimal solution for a given class hierarchy. We employ SPIN's search engine to perform the exhaustive search. In Algorithm 1 we present the pseudocode of our co-simulation approach. The following sections elaborate in more details on each of the stages of the framework.

3.1 Formal Verification

Every G&S implementation operates under the assumption that when a prime number is reused, it is assigned to non-conflicting classes. In addition, the maximum type ID must fit within the boundaries of a memory word. We check these invariants during the program verification phase. Establishing the validity of the G&S invariants is done by straightforward application of model-checking with SPIN. In SPIN the critical system properties are expressed in the syntax of linear time logic. Based on the G&S abstract specification, the model-checker performs a systematic exploration of all possible states. In case of failure, SPIN provides a counterexample that demonstrates a behavior that has led to an illegal state. In our model, the invariants are expressed as a *never claim* [Holzmann 2003], and are checked just before and after the execution of every statement.

3.2 Evaluation

SPIN has been previously employed to implement solutions of scheduling [Brinksma and Mader 2000] and discrete optimization [Ruys 2003] problems. The problem we face in the G&S heuristics is a combinatorial optimization problem [Nemhauser and Wolsey 1988]. Given a finite set I, a collection F of subsets of I, and a real-valued function w defined on I, a discrete optimization problem could be defined as the task of finding a member S of F, such that: $\sum_{e \in S} w(e)$ is as small (or as large) as possible.

Except for the simplest cases, a discrete optimization problem is difficult because its design space is typically disjoint and nonconvex. Therefore, the optimization methods applied to continuous optimization problems cannot be utilized in this case. In a small discrete problem, it would be possible to exhaustively list all possible combinations. As the number of parameters increase, the state explosion makes optimizations difficult. The two general strategies for approaching a discrete optimization problem can be classified as *deterministic* and *probabilistic*. What we do for the G&S exploration in SPIN could be described as applying a deterministic approach for the evaluation of a set of proposed probabilistic methods. The Branch and Bound method [Nemhauser and Wolsey 1988] guarantees the discovery of the global optimum in the cases when the problem is linear or convex and is the most frequently used discrete optimization

Table I. Enumeration of all solutions.

$id_c = k_c \times k_b \times (ka)^2$	k_a	k_b	k_c
60	2	3	5
60	2	5	3
90	3	2	5
90	3	5	2
150	5	2	3
150	5	3	2

method. It is based on the sequential analysis of the discrete tree of each parameter. The branches that can be estimated to reach invalid or unfeasible solutions are consequently eliminated. This simple optimization could also be applied in some limited cases in the SPIN's Fast Dynamic Casting exhaustive search. Let us explore a class hierarchy with three classes A, B, and C, where B is derived from A, and C is derived from both A and B. In this case, we have $S_{classes}=\{A, B, C\}$, $P=\{2, 3, 5\}$, and $id_{leaf}=\{id_c\}$. The enumeration is given in Table I. We assume that the computation starts at a state S_0 where all three keys k_a, k_b, and k_c are uninitialized. Then we assign possible values from the set P to the key variables of the classes A, B, and C. The enumeration shown above can be expressed as the computation shown on Figure 3. The graph shows only the valid states of the computation. There are a number of invalid states that are not shown on the graph. For example, according to the rules defined in G&S, it is possible to reuse some of the prime numbers in P. Thus, we can try and add an edge k_b=2 in state S_1, however the reuse of 2 in this case is invalid since A and B are conflicting classes.

The illustrated automation in Figure 3 provides a foundation for the construction of a Promela model for the deterministic solution. Each possible prime number assignment to a given class key is represented by a separate state transition in the exhaustive search model. SPIN initiates the optimum search at state S_0 and visits all possible states. At each end state the value of the minimum of the set of leaves, in this case represented only by id_c, is computed and compared to the current minimum. This approach is similar to the algorithm described by Ruys in [Ruys 2003] and shown in Algorithm 2. For such an application, we use the model checker in a somewhat unusual fashion. In this scenario, the validation property checks whether the value of id_c is greater than the current minimum. Each time this condition is

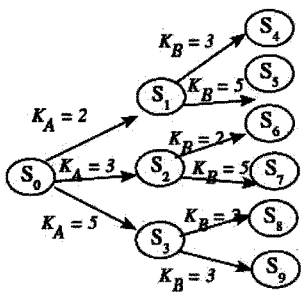


Figure 3. Exhaustive search computation.

Algorithm 2 Finding the global minimum in the state space.

1: *intput* : *Promela model M*
2: *output* : *the optimal minimum for the problem M*
3: $min = (worst\ case)\ maximum\ value\ for\ id$
4: **repeat**
5: *use SPIN to check M with condition* $(id_c > min)$
6: **if** *(error found)* **then**
7: $min = id_c$
8: **until** *(error found)*

violated, the current minimum is updated and the process is automatically repeated until SPIN confirms that there are no routes violating the specification. Since the solution is deterministic, it is guaranteed to discover the global optimum for type ID assignment. The performance of the G&S heuristics is measured by running a simulation of the G&S model that has been used earlier for verification. Now we are left with only one important task (not automated at this stage), the comparison of the results from the probabilistic and deterministic solutions. Once we identify a set of inconsistent results, we try to find a pattern and refine the G&S heuristics. Then the refined scheme is implemented in the probabilistic model and the evaluation process is reiterated.

3.3 Analysis

The simulation and enumeration models are continuously executed until, if possible, a set of instances with inconsistent solutions can be identified. Thus, each instance in the Set of Inconsistent Solutions (*SIS*), represents a given class hierarchy for which the deterministic and probabilistic approach has discovered different solutions. The class hierarchies for each test could be guided or created in a random fashion. For the generation of the test data in our experiments we implemented a pseudo random class hierarchy generation algorithm, in a manner similar to the TGFF (Task-Graphs-For-Free) method as described in [Dick et al. 1998]. We look for patterns among the collected hierarchies in *SIS* and seek clues that can lead us to improvements of the G&S scheme. Potential improvements are tested by adding them to the G&S model and evaluating their effect. Such scheme modifications are carefully selected since it is possible that they might enhance a given G&S feature and at the same time weaken another. Ideally, the improvements lead to a heuristic scheme that provides the best solutions for a larger number of the test hierarchies and at the same time has a time complexity equal to or less than the earlier heuristic scheme.

With the numerous advanced state space reduction techniques utilized by the SPIN model checker, little can be done to further optimize the exhaustive search. Class hierarchies of double or triple the size of the ones presented in the paper can possibly be facilitated with increased computational power and the parallelization of our approach. In the current framework, the exhaustive search is used to identify flaws in the G&S type ID assignment scheme. The goal of our experiments is to reach quick and effective optimization of the G&S scheme, and we have been able to achieve it with the current size of our models. A promising direction for our future research is to devise a parallelization scheme for our methodology, so that we can

perform simulations on a larger scale.

4. APPLICATION IN MISSION-CRITICAL SOFTWARE

Modern space mission systems have evolved from simple embedded devices into complex computing platforms with high autonomy and an exceptionally large demand for human-computer interaction. Consequently, such systems require reliable and flexible data systems managing the collection, storage, and transportation of data. The Mission Data System (MDS) is the Jet Propulsion Laboratory's state-and goal-oriented framework for building embedded control systems with a high degree of autonomy. MDS provides the building blocks for the implementation of embedded platforms based on the concepts of state estimation and control. The Data Management Services (DMS) is the MDS component responsible for the production, storage, processing, and transfer of control and scientific data. In [Wagner 2005] Wagner defines the challenges of data management in MDS as the problems of producing and storing data and converting the data to various formats as needed by its consumers. In addition, DMS needs to ensure the secure and lossless transport of the data with limited resources and through unreliable physical medium. To design and relate the data system entities, DMS employs concepts from high-level ISO C++ including templates, object-oriented class encapsulation, and dynamic casting necessary for the conversion of the data formats.

The actual telemetry data objects in MDS communicate with each other via byte streams produced by the transport protocol (e.g. spacecraft to ground communication). The receiver of the telemetry data needs to recreate the data object from the byte stream and thus invoke type casting in numerous occasions. Constant-time dynamic cast is also needed by the MDS Goal Network in the case when a controller or estimator [Wagner 2005] passes a goal via the Coordinating Goal Network(CGN), typically a large dynamic data structure. In CGN the goal is propagated using only its abstract attributes (start and end time, and the associated state variable). The achiever object who eventually picks up the goal needs to reconstruct the data object via dynamic downcasting to the specific type that conveys the state-specific achievement criteria. The application of the common compiler implementation of dynamic cast has proved to be unacceptable due to poor performance and the lack of the timing guarantees.

The G&S scheme was devised as a solution to a real industrial problem related to C++ use for hard real time code. Inquiries in the C++ community revealed that the problem was fundamental and common, rather than isolated: developers simulate dynamic casting with other language features, leading to type-unsafe special-purpose code or the avoidance of best object-oriented practices. Naturally, such workaround code slows down development, complicates maintenance, and increases the need for testing.

5. RESULTS

We applied the co-simulation process described in the previous section to a large number of class hierarchies. The tested hierarchies are not built into our models. Instead, we have applied a methodology reminiscent to TGFF [Dick et al. 1998] to

automatically generate hundreds of possible test cases. For illustration, we show the results from a set of seven pseudo random class hierarchies (Appendix A). The results of the G&S heuristics model and the exhaustive search are shown in Table II. A brief comparison of the results indicates that the G&S heuristics do not give optimal performance for class hierarchies with non-virtual multiple-inheritance. A closer look at the algorithm reveals that the priority calculation routine takes into account only the number of descendants that each class has. Let us consider the class hierarchy from test case 7. We notice that according to the current scheme, the base classes 0, 1, and 2 all get the same priority rank since they all share the descendant 6. Class 6 is at the lowest level of the hierarchy and has the largest number of ancestors. If we would like to optimize the heuristics, we must find a way to increase the priority of base class 2. Our reasoning is derived from the fact that Class 2 is ambiguous and the leaf Class 6 contains two copies of it. Similarly, let us have a closer look at test case 1. In the optimal solution, Class 5 takes the lower prime number (11) compared to Class 4, despite the fact that its only descendant has less ancestors compared to Class 4. The reason for this result is the fact that the derived Class 3 contains two ambiguous bases while Class 4 contains only one ambiguous base. As a result of our analysis we conclude that higher priority should be given to derived classes and their ancestors who contain more ambiguous base classes. To fix these weaknesses, we extend the G&S heuristics by adding two simple rules:

(1) We count every ambiguous ancestor twice when we determine the number of ancestors to each class
(2) For each base class, we count the number of derived classes that include more than one copy of it, and add that number directly to its priority

We call this enhanced G&S heuristics Fast Dynamic Casting Plus (FDC+). As Table

Table II. Co-simulation of the seven cases from Appendix A.

Case No	G&S	Exhaustive search	FDC+
Case 1 (keys)	(2, 3, 5, 7, 11, 13, 17)	(3, 2, 5, 7, 13, 11, 17)	(3, 2, 5, 7, 13, 11, 17)
Case 1(*ids* of all leaves)	(16380, 16830)	(13860, 13260)	(13860, 13260)
Case 2 (keys)	(2, 13, 3, 5, 17, 7, 11)	(2, 13, 3, 5, 17, 7, 11)	(2, 13, 3, 5, 17, 7, 11)
Case 2 (*ids* of all leaves)	(1326, 2310)	(1326, 2310)	(1326, 2310)
Case 3 (keys)	(2, 3, 13, 5, 7, 17, 11)	(2, 3, 13, 5, 7, 17, 11)	(2, 3, 13, 5, 7, 17, 11)
Case 3 (*ids* of all leaves)	(26, 51, 2310)	(26, 51, 2310)	(26, 51, 2310)
Case 4 (keys)	(2, 3, 5, 7, 11, 13, 17)	(2, 3, 5, 7, 11, 13, 17)	(2, 3, 5, 7, 11, 13, 17)
Case 4 (*ids* of all leaves)	(2310, 1547)	(2310, 1547)	(2310, 1547)
Case 5 (keys)	(2, 3, 5, 7, 11, 7, 11)	(2, 3, 5, 7, 11, 7, 11)	(2, 3, 5, 7, 11, 7, 11)
Case 5 (*ids* of all leaves)	(42, 66, 70, 110)	(42, 66, 70, 110)	(42, 66, 70, 110)
Case 6 (keys)	(2, 3, 5, 11, 13, 7, 17)	(2, 3, 5, 11, 13, 7, 17)	(2, 3, 5, 11, 13, 7, 17)
Case 6 (*ids* of all leaves)	(66, 78, 420, 170)	(66, 78, 420, 170)	(66, 78, 420, 170)
Case 7 (keys)	(2, 3, 5, 7, 11, 13, 17)	(3, 5, 2, 7, 11, 13, 17)	(3, 5, 2, 7, 11, 13, 17)
Case 7 (*ids* of all leaves)	(2552550)	(1021020)	(1021020)

II shows, for the initial set of test cases, FDC+ performance is 100% equivalent to the performance of the deterministic approach. In the performed tests, we have generated 127 pseudo random class hierarchies and applied G&S, FDC+, and the exhaustive search to each one of them. The experimental results showed that FDC+ was able to yield the best type ID assignment in 85% of the class hierarchies compared to 48% for the G&S heuristics. The time performance of the three schemes is shown in Figure 4. While the time performances of the G&S and FDC+ algorithms are equal and both run in a very low constant-time (the function at 00:01 min on Figure 4), logically the time performance of the exhaustive search increases exponentially with the increase of the number of classes nodes in a given class hierarchy. The analysis of the test results indicated that FDC+ finds a better type ID compared to the G&S approach in 39% of the test scenarios. For the greater part of the test cases, FDC+ matched the optimal type ID assignment computed by the exhaustive search. This efficiency boost is due to the optimized performance of FDC+ in the cases where multiple non-virtual inheritance is present in the class hierarchy. We have also observed that the implementation of these optimizations does not lead to efficiency loss in other scenarios and the performance of FDC+ is always at least as good as the performance of G&S. Our experimental results have indicated that the introduced optimizations in FDC+ have fixed a weakness of the original G&S approach and have improved the success rate in finding the best type ID assignment. The G&S scheme requires a key of a memory size that is a function of the size and shape of a class hierarchy. Thus, the improved heuristics almost double the size of class hierarchies that can be handled by a given key size. Since the scheme gets significantly slower when a key gets too large for a machine word, the improvements to the heuristics address the main limitation of the G&S scheme.

6. BASIC QUERY: EXTRACTING SEMANTIC INFORMATION FROM CODE

The heavy computational overhead of the model-checking tools as well as the problem

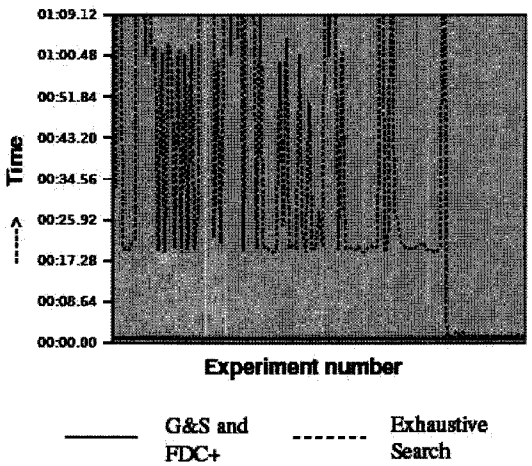


Figure 4. Search time for type ID assignment.

of state space explosion limits the applicability of our SPIN models to the process of system simulation and analysis. For its use in practice, the FDC+ scheme needs to provide a compile time guarantee for each program that all of its type IDs fit within the allowed bounds of a memory word. To achieve a practical and verifiable fast dynamic casting operation we enhanced our implementation with a static analysis module that checks the type ID assignment's validity.

In the remaining part of this section we describe the design and implementation of Basic Query (BQ), an innovative library for extracting semantic information from C++ source code. BQ user-defined actions are executed by traversing a compact high-level abstract syntax tree (AST) called Internal Program Representation (IPR). IPR is at the center of a C++ static analysis framework named The Pivot [Stroustrup and Reis 2005]. We take advantage of BQ's simplicity and efficiency in formulating and combining static analysis queries to construct a set of graphs representing all class hierarchies in a C++ program. Having the class graphs at compile time allows FDC+ to guarantee, prior to the program's execution, that all assigned type IDs fit within the required bounds of a 64-bit memory word.

The Pivot is a compiler-independent platform for static analysis and semantics-based transformation of the complete ISO C++ programming language and some advanced language features proposed for the next generation C++, C++0x [Becker 2006]. The Pivot represents C++ programs in two distinct formats (Figure 5):

(1) Internal Program Representation (IPR). IPR is a high level, compact, fully typed abstract syntax tree that can represent complete ISO C++ programs as well as incomplete program fragments and individual translation units
(2) eXternal Program Representation (XPR). XPR is a persistent and human readable format for program representation. XPR uses a prefix notation and is quick to parse (a single token look ahead and no symbol table needed)

Fundamental to our BQ library is the design of a fast and flexible methodology for

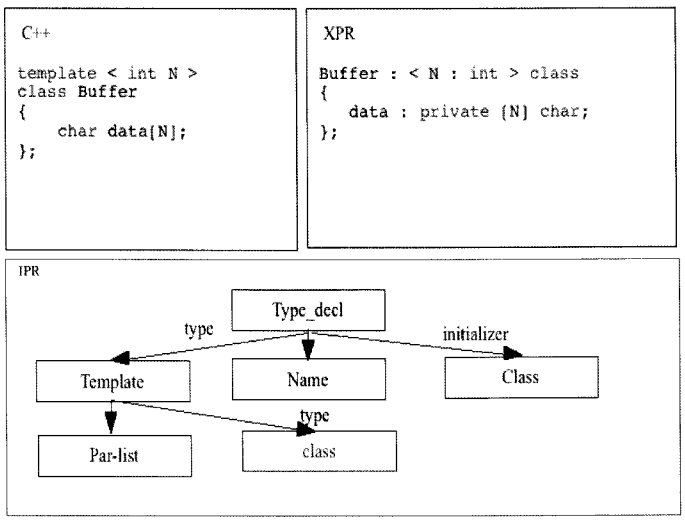


Figure 5. An XPR and IPR representations of a C++ template class definition.

traversing the IPR, The Pivot's AST. We define a depth-first search (DFS) visitor class, called the IPR Xplorer Visitor Class, that performs the AST search following the order of the ISO C++ grammar definition [Stroustrup 2000]. The Xplorer allows the programmer to statically define a set of actions to be executed during the DFS traversal including a terminating condition as well as actions upon the encounter of specific IPR nodes (C++ expressions, declarations, and statements) and AST edges (interfaces of the IPR nodes). In such a way, the cost of a user-defined action could be less than a single traversal of the abstract syntax tree. The functionalities and user interfaces of the Xplorer visitor are reminiscent to the syntax and operation of the Boost's DFS Visitor [Abrahams and Gurtovoy 2004]. When an action is specified, the programmer instantiates each of these classes with two compile-time arguments, a *TP* (trigger point), identifying the exact point of triggering the action, and a *TN* (target node), specifying the type of IPR nodes which are the traversal's target. The following examples illustrate the usage of the Xplorer visitor: *xplore_expr_node <discover, ipr::Call>*, we specify an action at the point of discovery of each *ipr::Call* node, and *xplore_stmt_node <body, ipr::Switch>*, a user-defined action is executed prior to exploring the edge *body* of an IPR node of type *ipr::Switch*.

In some scenarios it is preferred to have linear access to the nodes of a program unit and at the same time manipulate the AST through an intuitive and familiar user interface. Our Xplorer Visitor defines the classes: *I P R_Visitor* and *I P R_Iterator*. Their design closely follows the functionality and philosophy of the visitor design pattern [Abrahams and Gurtovoy 2004] and the C++ STL Iterator [Stroustrup 2000] classes, providing a convenient way to search, manipulate, or modify a set of IPR objects. The convenience of this method comes at a certain price: the DFS traversal needs to collect and store in advance all of the nodes from a program unit, thus the cost of the user-specified actions is at least a single traversal of the AST.

BQ user-defined actions are constructed at compile time by using the mechanism of expression templates. An expression template is a programming technique that relies on the compiler's evaluation of template arguments in order to pass C++ expressions as inlined function arguments. This approach has been successfully applied in a number of Boost Libraries [Abrahams and Gurtovoy 2004] to deliver efficient and modern C++ designs that avoid the use of costly C-style pointers to callback functions. In the case of BQ queries, we employ expression templates to evaluate at compile-time a combination of user-specified parameters and pass the user's request for run-time execution as an inlined function argument. Thus, by eliminating the necessity to resort to an object-oriented design utilizing pointers to class member function to specify user intent, we gain performance and flexibility. Expression templates are not used in the construction of the entire pattern tree because of the heavy syntax and the reduced expressiveness that such an approach would impose. Instead, the 'glue' between all statically computed *BQ elements* is encoded in the *BQ operations* (Table III). The clean and flexible syntax of the BQ user-defined actions is achieved through the exploitation of the C++ compiler's ability to perform complex template argument inference.

A *BQ action* (also a BQ pattern) consists of three components: a *Recursive Query Object* (*RQO*) containing the root of the traversal as well as the result from an

applied pattern or a sequence of patterns, a set of *BQ elements*, and a set of *BQ operations*. At each step of the AST traversal, the RQO decides whether the target is reachable from the current point and carry on with the execution of the pattern or terminate the search. A *BQ pattern* is expressed through a combination of a number of BQ elements and BQ operations applied to the recursive query object. There are a number of possible applications of the BQ operations on the BQ elements (Table IV). A BQ element specifies one or several edges in the pattern tree. A BQ element could be one of three possible types:

(1) *Exe_member*<x, e>. (EM) generates a straightforward edge e from an IPR node x. For example, if the vertex x is an IPR node of type *ipr*:: *Type_decl* and the edge e is *ipr*:: *initializer*, the result of the operation is the IPR node yielded by the execution of the IPR interface x-> *initializer* (that is the initializer of a C++ type declaration).
(2) *Exe_condition*<x, e, c>. (EC) generates an edge e from an IPR node x, only if a specified boolean condition c is met
(3) *Exe_iprseq*<x, e_n>. (ES) produces a sequence of edges e_n resulting in a set of IPR nodes. An example of such an edge in the pattern tree is the call to retrieve all bases of a class declaration (x-> *bases*( )).

An important component of our class hierarchy extraction routine is the specification of a user-defined action searching for all class declarations in a program that are children of a certain base class. To do that we specify a BQ check that tests every pair of classes for a parent-child relationship. As an example of a BQ routine, we

Table III. BQ operations.

Operation	Operand	Description
Apply	<	execute an action specified by a BQ element
Apply and Evaluate	∧	executes a BQ element and returns the result (a bool, an IPR node or a set of nodes)
Evaluate	->	returns the result from the application of a BQ pattern

Table IV. Application of the BQ operations.

Operation	Result	Operation Description	Result Description
RCO < *ES*	*Set of IP R Nodes*	applies an ES	sequence of IPR nodes (such as a list of base classes)
RCO < *EM*	*RCO*	executes an EM, stores the result in RQO	a pointer to RQO
RCO < *EC*	*RCO*	executes an EC, stores the result in RQO	a pointer to RQO
RCO ∧ *EC*	*bool*	executes an EC, stores the result in RQO	the evaluation of EC's condition
(Set of IP R Nodes) ∧ *EC*	*bool*	searches for a match for EC's condition	true if at least one instance satisfies the predicate

present the definition of the function object *Is_derived_from* (Algorithms 3 and 4), that tests a pair of class nodes for a parent-child relationship. In this section we presented the design and application of the static analysis tools that help us deliver a practical and verifiable fast dynamic cast implementation. The remaining algorithms from our class graph construction routine (that we do not show in this paper) are a technical detail of simply applying the discussed techniques.

Algorithm 3 Testing a pair of C++ classes for a parent-child relationship

1: $RCO : ipr :: Expr$
2: $EC1 : ipr :: Type\_decl- > has\_initializer$
3: $EM1 : ipr :: Type\_decl- > initializer$
4: $ES1 : ipr :: Class- > bases* - > ipr :: Base\_type$
5: $EC2 : ipr :: Base\_type- > name\_cmp$
6: $Is\_derived\_from : RC < EC1 < EM1 < ES1 \wedge EC2- > bool$

Algorithm 4 Testing a pair of C++ classes for a parent-child relationship, source code

1: Input: an IPR Expression node e
2: $Recursive\_query\ RCO(e)$;
3: $Exe\_condition < ipr :: Type\_decl, has\_initializer, bool > Has\_Init(\&val\_cmp < bool >, true)$;
4: $Exe\_member < ipr :: Type\_decl, initializer > Init$;
5: $Exe\_iprseq < ipr :: Class, ipr :: Base\_type, bases > Get\_Bases$;
6: $Exe\_condition < ipr :: Base\_type, name, constipr :: Name\&, std :: string > Is\_Name(\&name\_cmp, parent\_name)$;
7: $return\ RCO < Has\_Init < Init < Get\_Bases \wedge Is\_Name$;

7. CONCLUSION

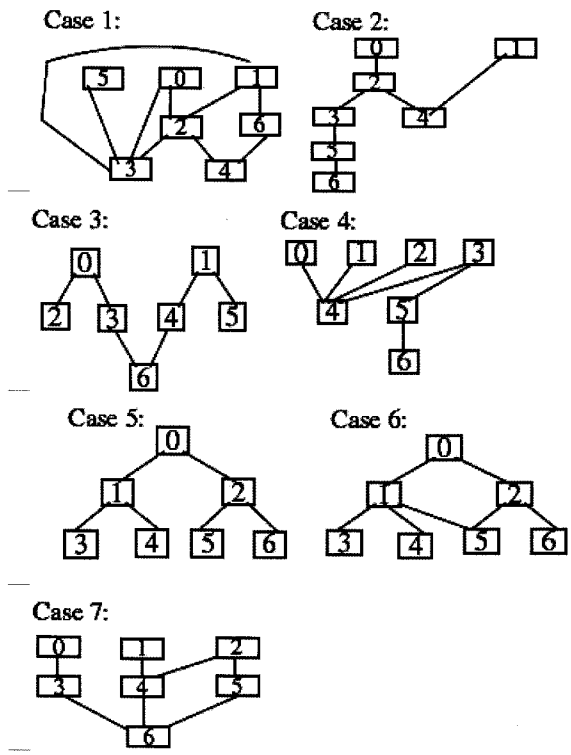
In this work we applied co-simulation of the deterministic and probabilistic solutions to the combinatorial optimization problem posed by the G&S type ID assignment scheme. Our framework proved successful in verifying and refining the existing G&S heuristics. We demonstrated how we use the simulation results to devise improvements to the G&S algorithm and evaluate them. The results from our experiments indicate that the improved G&S heuristics (FDC+) provide the optimal type ID assignment in 85% of the class hierarchies, compared to 48% for the regular G&S algorithm. The efficiency of the type ID assignment scheme has significant importance for the performance of the fast dynamic casting by Gibbs and Stroustrup [Gibbs and Stroustrup 2006]. This paper presented a practical approach of how to discover improvements to the type ID assignment scheme in a simple and effective manner. The main advantage of the presented approach is the ease and simplicity of the discovery and test for potential improvements. The improved heuristics that we have described in this work almost doubles the size of class hierarchies that can be handled by a given key size. A more extensive simulation might suggest further improvements to the type ID assignment scheme. Our main goal in this work has been to demonstrated how an algorithm optimization problem encountered has been successfully automated and moreover that its automation has led us to quick but significant improvements of the initial scheme. In addition, we introduced the design and application of Basic Query, an innovative expression-template based library for extracting semantic

information from C++ source code. We demonstrated how to apply Basic Query to achieve a practical and verifiable implementation of the FDC+ scheme.

8. ACKNOWLEDGEMENTS

We would like thank David Wagner and Kirk Reinholz from the Jet Propulsion Laboratory and Peter Pirkelbauer from Texas A&M University for the meaningful discussions on work. We are grateful to the anonymous referees for their insightful comments and suggestions.

A. APPENDIX



REFERENCES

ABRAHAMS, D. AND GURTOVOY, A. 2004. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional.

BECKER, P. 2006. Working Draft, Standard for Programming Language C++, ISO WG21 N2009.

BRINKSMA, E. AND MADER, A. 2000. Verification and Optimization of a PLC Control Schedule. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. Springer-Verlag, London, UK, 73–92.

DICK, R. P., RHODES, D. L., AND WOLF, W. 1998. TGFF: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*. IEEE Computer Society, Washington, DC, USA, 97–101.

GIBBS, M. AND STROUSTRUP, B. 2006. Fast dynamic casting. *Softw. Pract. Exper. 36*, 2, 139–156.

GLUCK, R. AND HOLZMANN, G. 2002. Using spin model checker for flight software verification. In *In Proceedings of the 2002 IEEE Aerospace Conference.*

GOLDTHWAITE, L. 2006. Technical Report on C++ Performance. In *ISO/IEC PDTR 18015.*

HOLZMANN, G. 2003. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, Massachusetts.

INGHAM, M., RASMUSSEN, R., BENNETT, M., AND MONCADA, A. 2004. Engineering Complex Embedded Systems with State Analysis and the Mission Data System. In *In Proceedings of First AIAA Intelligent Systems Technical Conference 2004.*

ISO/IEC 14882 INTERNATIONAL STANDARD. 1998. *Programming languages C++.* American National Standards Institute.

LERNER, S., MILLSTEIN, T., AND CHAMBERS, C. 2003. Automatically proving the correctness of compiler optimizations. In *PLDI 03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation.* ACM Press, New York, NY, USA, 220–231.

LOWRY, M. R. 2002. Software Construction and Analysis Tools for Future Space Missions. In *TACAS* (2002-03-18), J.-P. Katoen and P. Stevens, Eds. Lecture Notes in Computer Science, vol. 2280. Springer, 1–19.

MUSUVATHI, M. AND ENGLER, D. R. 2004. Model checking large network protocol implementations. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation.* USENIX Association, Berkeley, CA, USA, 12–12.

NEMHAUSER, G. L. AND WOLSEY, L. A. 1988. *Integer and combinatorial optimization.* Wiley-Interscience, New York, NY, USA.

PERROW, C. 1999. *Normal Accidents.* Princeton University Press.

RTCA. 1992. Software Considerations in Airborne Systems and Equipment Certification (DO178B).

RUYS, T. C. 2003. Optimal scheduling using branch and bound with spin 4.0. In *Proceedings of the 10th International SPIN Workshop on Model Checking software*, T. Ball and S. K. Rajamani, Eds. Lecture notes in Computer Science, vol. 2648. Springer Verlag, Berlin, 1–17.

SCHUMANN, J. AND VISSER, W. 2006. Autonomy Software: V&V Challenges and Characteristics. In *Proceedings of the 2006 IEEE Aerospace Conference.*

STOICA, A., KEYMEULEN, D., CSASZAR, A., GAN, Q., HIDALGO, T., MOORE, J., NEWTON, J., SANDOVAL, S., AND XU, J. 2005. Humanoids for lunar and planetary surface operations. In *Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics.*

STROUSTRUP, B. 2000. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

STROUSTRUP, B. 2004. Abstraction and the c++ machine model. In *ICESS* (2005-09-14). Lecture Notes in Computer Science, vol. 3605. Springer, 1–13.

STROUSTRUP, B. AND REIS, G. D. 2005. Supporting SELL for High-Performance Computing. In *In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, LCPC 2005.*

VELDHUIZEN, T. L. 1995. Expression templates. *C++ Report 7*, 5 (June), 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

VOLPE, R. 2005. Rover Technology Development and Mission Infusion Beyond Mars Exploration Rover. In *IEEE Aerospace Conference.*

WAGNER, D. 2005. Data Management in the Mission Data System. *In Proceedings of the IEEE System, Man, and Cybernetics Conference.*

Damian Dechev is a graduate student and PhD candidate at the Department of Computer Science at Texas A&M University at College Station, Texas, USA. His research interests are in the areas of programming techniques and tools, practical nonblocking synchronization, and program modeling and verification. Damian Dechev is a member of the Parasol Research Lab and conducts his research under the supervision of Dr. Bjarne Stroustrup. Dechev has performed joint research as a research intern with the Jet Propulsion Laboratory's Mission Data System Project in 2005 and 2006. He has been awarded with the Adobe IAP Scholarship. Dechev holds an MS in Computer Science from the University of Delaware and a BS in Computer Science from the University of Indianapolis.

Rabi Mahapatra is an associate professor of Computer Science, Texas A&M University, at College Station, USA. His research interests include embedded systems, system on chip, VLSI architecture, networking and cyber physical systems. Mahapatra received PhD from Indian Institute of Technology, India. He is currently directing the Embedded System Research Group at Texas A&M University. Mahapatra is a Ford Fellow and Distinguished visitor of IEEE Computer Society. His research is funded by Federal agencies like NSF, FAA, NASA, and industries like IBM, Boeing, BAE, Honeywell, GE Avionics, etc. He has published more than 100 research articles in referred International journals and conferences. He can be contacted at HYPERLINK "mailto:rabi@cs.tamu. edu"

Bjarne Stroustrup is the designer and original implementer of C++ and the author of The C++ Programming Language, The Annotated C++ Reference Manual, and The Design and Evolution of C++. Having previously worked at Bell Labs and AT&T Labs - Research, he currently is the College of Engineering Chair in Computer Science Professor at Texas A&M University. The recipient of numerous honors, including the Dr. Dobb's Excellence in Programming Award (2008), Dr. Stroustrup is a member of the National Academy of Engineering, an AT&T Fellow, an AT&T Bell Laboratories Fellow, an IEEE Fellow, and an ACM Fellow. His research interests include distributed systems, programming techniques, software development tools, and programming languages. Dr. Stroustrup holds an advanced degree from the University of Aarhus in his native Denmark and a Ph.D. in Computer Science from Cambridge University, England.