Object Model Construction for Inheritance in C++ and Its Applications to Program Analysis

Conference Paper · March 2012

DOI: 10.1007/978-3-642-28652-0_8

Sriram Sankaranarayanan
University of Colorado Boulder

Object Model Construction for Inheritance in C++ and its Applications to Program Analysis

Jing Yang[1,2], Gogul Balakrishnan[1], Naoto Maeda[3], Franjo Ivančić[1], Aarti Gupta[1],
Nishant Sinha[⋆4], Sriram Sankaranarayanan[5], and Naveen Sharma[6]

[1]NEC Labs America [2]University of Virginia [3]NEC Corporation, Japan
[4]IBM Research, India [5]University of Colorado, Boulder [6]NEC-HCL Tech., India

Abstract. Modern object-oriented programming languages such as C++ provide convenient abstractions and data encapsulation mechanisms for software developers. However, these features also complicate testing and static analysis of programs that utilize object-oriented programming concepts. In particular, the C++ language exhibits features such as multiple inheritance, static and dynamic typecasting that make static analyzers for C++ quite hard to implement. In this paper, we present an approach where static analysis is performed by lowering the original C++ program into a semantically equivalent C program. However, unlike existing translation mechanisms that utilize complex pointer arithmetic operations, virtual-base offsets, virtual-function pointer tables, and calls to run-time libraries to model C++ features, our translation is targeted towards making static program analyzers for C++ easier to write and provide more precise results. We have implemented our ideas in a framework for C++ called CILpp that is analogous to the popular C Intermediate Language (CIL) framework. We evaluate the effectiveness of our translation in a bug finding tool that uses abstract interpretation and model checking. The bug finding tool uncovered several previously unknown bugs in C++ open source projects.

1 Introduction

Modern object-oriented programming languages provide convenient abstraction and data encapsulation mechanisms for software developers. Such mechanisms include function and operator overloading, constructors and destructors, multiple class inheritance, dynamic virtual-function dispatch, templates, exceptions, functors, standard libraries such as STL and BOOST. However, on the flip side, these features complicate the static analysis of programs that use such features. In the past decade, there have been numerous approaches for static program analysis techniques based on source code. These tools rely on abstract interpretation [12] or software model checking [9], such as ASTREÉ [13], Saturn [40], SLAM [3], CBMC [8], Java PathFinder [16], and FindBugs [17]. However, in practice, these tools have largely been developed and optimized to either address C or Java. With respect to static program analysis techniques for object-oriented source code, most work has addressed Java partly because of its less intricate class hierarchy concept. However, in industry, C++ is one of the predominant development languages. Due to its intrinsic complexity mixing object-oriented programming

[⋆] Work done while at NEC Labs America.

on top of full-fledged C code, there is a stark need for static program analysis for C++ supporting all of its many features.

There are several popular front-ends, such as Comeau C/C++ [10], EDG [15], LLVM [19], and ROSE [27], that support all the complex features of C++. In spite of the availability of C++ front-ends, it is still hard to perform static analysis of C++ programs as observed by the developers of Clang [6, 7]:

> *" . . . Support in the frontend for C++ language features, however, does not automatically translate into support for those features in the static analyzer. Language features need to be specifically modeled in the static analyzer so their semantics can be properly analyzed. Support for analyzing C++ and Objective-C++ files is currently extremely limited, . . . "*

Modeling the C++ features directly in static analysis is a non-trivial task because the C++ semantics is quite complicated [32]. Moreover, every static analyzer may need to encode the semantics differently depending upon the requirements of the analysis. For complex analysis, this process can easily get out of hand. One particular issue in handling of C++ programs compared to other object-oriented programming languages such as Java is the complexity of allowed class hierarchies. C++ allows multiple inheritance, where a class may inherit from more than one class. Presence of multiple inheritance gives rise to complex class hierarchies which must be handled directly (and precisely) by any static analysis. Furthermore, multiple inheritance complicates the semantics of otherwise simple operations such as casts and field accesses [33]. Therefore, techniques developed for Java are not readily applicable to C++ programs. It is important to emphasize that multiple inheritance is used quite frequently by developers even in large C++ projects. Nokia's cross-platform UI framework Qt [25], Apache's Xerces project [1], and g++ standard IO stream library are good examples.

An alternative approach to analyzing C++ programs with complex hierarchies is to utilize compiler front-ends that compile C++ programs into equivalent C programs (often referred to as "lowering of C++ programs"). A number of such approaches exist, starting from the earliest C++ compilers, such as Cfront [34]. Today, there are commercial C++ front-ends available that provide similar features, for example Comeau C/C++ [10] or EDG [15]. Such translations are generally geared towards runtime performance and small memory footprint. Therefore, these front-ends heavily utilize pointer arithmetic operations, virtual-base offsets, virtual-function pointer tables, and rely on runtime libraries, to achieve those goals. Such translations are hardly amenable to precise program analysis.

Consider, for example, the translation using one such generic lowering mechanism shown in the middle column of the table in Fig. 1. Note how a static_cast operation in row (a) of Fig. 1 is translated into an adjustment of the source pointer pr by 8 bytes in the lowered C program. Similarly, the dynamic_cast operation is translated into a call to an opaque runtime function _dynamic_cast, which may change the source pointer as in the case of static_cast. Such pointer adjustment operations are crucial for preserving the semantics of class member accesses in the lowered C program. However, static analysis algorithms typically assume that such pointer adjustments (which are akin to pointer arithmetic operations) do not change the behavior of the program, and therefore, may ignore them completely. Consequently, the analysis of the lowered

	Standard C++-to-C lowering	CHROME-based lowering

(a)	`pb =`
	`((struct B *) ((pr != ((struct R *) 0))`
	`? (((char *)pr) - 8UL) : ((char *) 0)));`
		`assert(pr->soid == B_R);`
		`pb = pr ? pr->dervB : 0;`

(b)	`pl = ((pr != ((struct R *)0))`
	` ? ((struct L *)(_dynamic_cast(...)))`
	` : ((struct L *)0));`
		`assert(pr->soid == B_R);`
		`pl = pr? pr->dervB->baseL : 0;`

(c)	`// Access virtual function table`
	`_T31 = ((((pl->_b_R)._vptr)) + 1);`
	
	`// Call virtual function`
	`(((void (*)(struct R * const))`
	`((_T31->f))) (((struct R *)`
	`(((char *) (&pl->_b_R)) +`
	`((_T31->d))))));`
		`switch(pl->soid) {`
		`case B_L:`
		`case L:`
		`{ T* prt = pl->baseT;`
		`prt->T::vfunc(prt);`
		`break;`
		`}`
		`default: assert(false);`
		`}`

(d)	`((*((struct T *) (((char *) pb) +`
	`((((pb->_b_1L)._vptr))[(-3)])))).tp)`
	`= ((int *) 0);`
		`pb->baseL->baseT->tp`
		`= (int *)0;`

Fig. 1: Comparison of the standard C++-to-C lowering and CHROME-based lowering mechanisms: (a) `pb = static_cast<B*>(pr)`, (b) `pl = dynamic_cast<L*>(pr)`, (c) `pl->vfunc()`, `vfunc` is a virtual function), and (d) `pb->L::tp = 0`, `L::tp` is a field of a shared base class).

C program is unsound. Alternatively, a conservative treatment of pointer arithmetic operations results in a large number of false positives, thereby reducing the usefulness of the analysis.

Further, note how a virtual-function call in Fig. 1(c) is translated into a complex combination of virtual-function pointer table lookups (via *vptr* variable) and unintelligible field accesses. Furthermore, in row (d) of Fig. 1, even a simple access to a field of a shared base class (see Sect. 2) is translated into an access through the virtual-function table [33]. Most conventional static analysis are not well equipped to precisely reason about such code. In particular, they are generally imprecise in the presence of arrays of function pointers or pointer offsets (such as the virtual-function pointer table). At a virtual-function call, a naive analysis may enumerate all potential callees, thus causing blowup in the computed call graph due to redundant function invocations. Alternatively, many techniques safely approximate arrays using summary variables, which leads to a severe imprecision in the resolution of field accesses and virtual-function calls.

Our Approach. In this paper, instead of encoding complex C++ semantics in static analyzers or simulating the behavior of the compiler, we adopt a more pragmatic approach to analyzing C++ programs. We believe that in order to allow for a scalable, yet precise, static analysis of C++, such techniques need not have to reason about the low-level physical memory layout of objects. Rather, we require a higher level of abstraction of the memory that is closer to the developer's understanding of the program.

Towards this goal, we propose a representation for modeling C++ objects that is different from the object layout representations used by a compiler. The alternative representation is referred to as CHROME (Class Hierarchy Representation Object Model Extension) and uses the algebraic theory of sub-objects proposed by Rossie and Friedman [30]. Using the CHROME object model, we employ a sequence of source-to-source transformations that translate the given C++ program *with* inheritance into a semantically equivalent C++ program *without* inheritance. Our source-to-source transformations comprises two main steps. The first step employs a *clarifier* module which makes implicit C++ features explicit. An example of such an implicit feature is the invocation of constructors, destructors, and overloaded operators. The second step involves the elimination of inheritance-related features using the proposed CHROME model. The translations are semantics-preserving and *static program analysis friendly*. Our aim is to allow state-of-the-art static program analyzers, that are currently oblivious to inheritance and multiple inheritance in programs, to naturally handle such transformed programs while maintaining their efficiency and precision.

The last column in Fig. 1 shows the CHROME-based transformations. The key idea underlying the CHROME model is to treat sub-objects due to inheritance as separate memory regions that are linked to each other via additional base class and derived class pointer fields. Instead of utilizing virtual-function pointer table lookups and address offset computations to resolve issues related to dynamic dispatch and casts, we instead follow a path in a sub-object graph utilizing these additional pointers to find the required sub-object of interest. Note that this sub-object graph walk may require multiple pointer indirections and would thus be inefficient for runtime performance as well as less memory efficient. However, static program analyzers routinely reason about heaps and pointer indirections. Hence, we build upon the strengths of these tools to allow for a efficient and precise analysis of complex C++ programs. The analysis of the resulting C++ program is also simplified because program analysis tools can treat casts and accesses to fields of inherited classes and virtual-function calls in the same way as regulars casts and accesses to fields of regular classes and normal function calls, respectively.

Our approach has multiple advantages. Various analyses can now focus on the reduced subset of C++ in a uniform manner without being burdened with having to deal with inheritance. Further, we may adopt simpler object-oriented analyses, e.g., those developed for Java programs, to analyze C++ programs. Finally, because the reduced subset is quite close to C, the given C++ program can also be lowered to C. This enables reuse of standard static analyses for C programs without necessitating the loss of high-level data representation.

The transformed source code can be potentially used not only for static analysis but also for dynamic program analysis, code understanding, re-engineering, runtime monitoring, and so on. We believe that our approach provides a uniform way to resolve the principal barrier to analyzing C++ programs, i.e., handling of inheritance precisely in a scalable fashion.

To illustrate the practical utility of our approach, we experiment with an in-house bug-finding tool called F-SOFT [18] that uses abstract interpretation [12] and model checking [9]. Our experiments show that our method provides significant benefits as compared to a straightforward C++-to-C lowering-based approach. We found a total of ten previously unknown bugs on some C++ open source projects, of which *five* were only found due to the precise modeling of objects using the CHROME object model. We also experimented with a publicly available bug finding tool called CBMC [8], which showed similar improvements when using the CHROME object model.

Contributions The key contributions of our paper are as follows:

- The CHROME (Class Hierarchy Representation Object Model Extension) object model representation for modeling objects of derived classes.
- An algorithm for translating a C++ program *with* inheritance into a semantically equivalent C++ program *without* inheritance using the CHROME object model.
- Performing the translation without the use of pointer arithmetic operations, virtual-base offsets, virtual-function tables, and runtime functions. Our source-to-source translation is designed with program analysis in mind and not for runtime performance or a small memory footprint.
- Illustrating the practicality of our approach by evaluating the effectiveness of the lowered C++ programs for abstract interpretation and model checking.
- A framework for C++ called `CILpp` that can be used to build analysis and verification tools for C++ programs. `CILpp` is analogous to the popular C Intermediate Language (CIL) framework for C programs [24].

Our implementation is based on the `EDG` frontend [15]. This allows us to focus on the analysis while being able to handle arbitrary C/C++ dialects. Our current lowering mechanism using the CHROME object model relies on the assumption that the tool chain is aware of the complete class hierarchy. This restriction simplifies the object model generation substantially since our generated object models do not have to address dynamic class loading, for example.

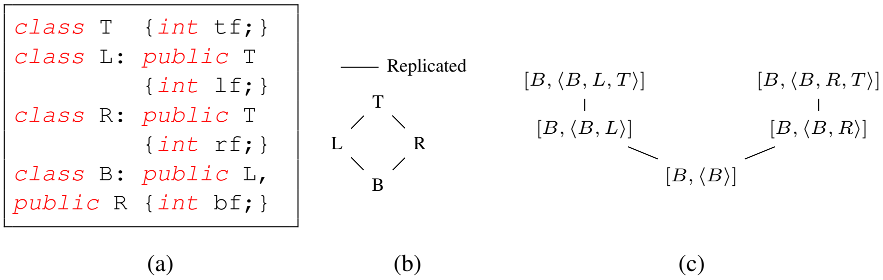
The rest of the paper is organized as follows. Sect. 2 describes the algebraic theory of Rossie-Friedman sub-objects. Sect. 3 presents the clarifier module that makes implicit C++ features explicit. Sect. 4 presents the CHROME object model and the source-to-source transformations that eliminate inheritance. Sect. 5 describes the results of our experiments. Sect. 6 discusses the related work. Sect. 7 concludes the paper.

2 Rossie-Friedman Sub-objects

Informally, the Rossie-Friedman sub-object model [30] is an abstract representation of the object layout. When a class inherits from another class, conceptually the base class is embedded into the derived class. Therefore, an object of a derived class consists of different (possibly overlapping) components that correspond to the direct and transitive base classes of the derived class. Intuitively, a *sub-object* refers to a component of a direct or transitive base class that is embedded in a derived class object. The complete derived class object is also considered to be a sub-object.

Example 1. Consider a class `L` that inherits from another class `T`. An object of type `L` consists of two sub-objects: (1) a sub-object of type `T` corresponding to the base class `T`, and (2) a sub-object of type `L` that corresponds to the complete object itself. Fig. 4(I)(a) shows the sub-objects of `L`. ■

C++ supports multiple inheritance through which a class inherits from more than one base class. In case of single inheritance, there is only one copy of every (direct or transitive) base class in a derived class object. However, with multiple inheritance, the number of sub-objects corresponding to a direct or transitive base class depends upon the number of paths between the base class and the derived class in the class hierarchy.



```
class T  {int tf;}
class L: public T
         {int lf;}
class R: public T
         {int rf;}
class B: public L,
public R {int bf;}
```

Replicated
T
L
R
B
$[B, \langle B, L, T\rangle]$
$[B, \langle B, R, T\rangle]$
$[B, \langle B, L\rangle]$
$[B, \langle B, R\rangle]$
$[B, \langle B\rangle]$
(a)
(b)
(c)

Fig. 2: Replicated multiple inheritance: (a) C++ program, (b) class-hierarchy graph, and (c) sub-object poset for class B.

Example 2. Consider the program shown in Fig. 2(a). Fig. 4(I)(b) shows the object layout for B. Because B inherits from L and R, a B-object contains sub-objects of type L and R. Further, the sub-objects of type L and R each contain a distinct sub-object of type T. Therefore, a B-object has two distinct sub-objects of type T: one inherited from class L and the other inherited from class R. ■

Virtual base classes It is often not desirable to have multiple sub-objects of a base class in a derived class. Therefore, to prevent replication of base class sub-objects in a derived class, C++ provides *virtual base classes*. Unlike a non-virtual base class, a sub-object of a virtual base class type is shared among the sub-objects of all its direct and transitive derived classes.

Example 3. Consider the class hierarchy in Fig. 3(a). The keyword *virtual* indicates that class T is a virtual base class. Fig. 4(I)(c) shows the object layout for B. A B-object contains sub-objects of type L and R as usual. However, because T is a virtual base class, it is shared among the direct and transitive derived classes L, R, and B. Therefore, a B-object contains only one sub-object of type T.

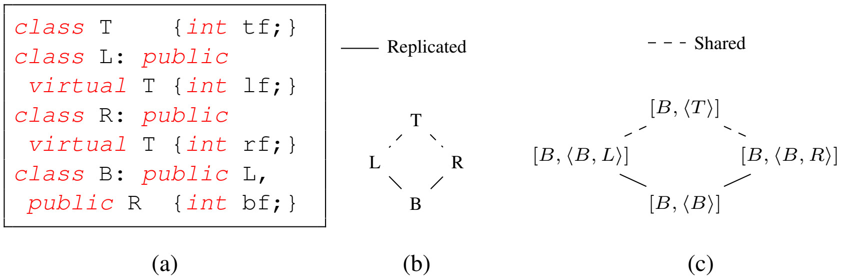
When a class inherits from a non-virtual base class, it is referred to as *replicated* inheritance. When a class inherits from a virtual base class, it is referred to as *shared* inheritance. The class hierarchy graph captures the shared and replicated inheritance relationships among the classes.

Definition 1 (Class Hierarchy Graph (CHG)**).** *A class hierarchy graph $\mathcal{G}$ is a tuple $\langle \mathcal{C}, \prec_s, \prec_r\rangle$, where $\mathcal{C}$ is the set of class names, $\prec_s \subseteq \mathcal{C} \times \mathcal{C}$ are shared inheritance edges, and $\prec_r \subseteq \mathcal{C} \times \mathcal{C}$ are replicated inheritance edges. Let $\prec_{sr} = (\prec_s \cup \prec_r)$.*

The formal description relies on the following operators to model transitive applications of $\prec_s$ and $\prec_r$. $<_s = (\prec_s)^+$, $\leq_s = (\prec_s)^*$, $<_r = (\prec_r)^+$, and $\leq_r = (\prec_r)^*$. Similarly, $<_{sr} = (\prec_{sr})^+$, and $\leq_{sr} = (\prec_{sr})^*$.

We require that the reflexive and transitive closure $\leq_{sr}$ of $\prec_{sr}$ is antisymmetric. This ensures that a CHG *is acyclic. Note that $(\mathcal{C}, \leq_{sr})$ is a poset.*

The Rossie-Friedman sub-object model formalizes the notion that, given a derived class D, a sub-object of a D-object is either the complete D-object or a component of a base



```
class T     {int tf;}
class L: public
 virtual T {int lf;}
class R: public
 virtual T {int rf;}
class B: public L,
 public R  {int bf;}
```

Fig. 3: Shared multiple inheritance: (a) C++ program, (b) class hierarchy graph, and (c) sub-object poset for class B.

class type B that is embedded in the D-object. Because C++ allows multiple inheritance, a D-object may have multiple sub-objects of a base class type B. Therefore, it is not sufficient to represent a sub-object of type B in a D-object simply as a pair $\langle \mathtt{D}, \mathtt{B} \rangle$. Rossie and Friedman distinguish different sub-objects of the same base class in a derived class using the path from the base class to the derived class in the CHG.

Definition 2 (Sub-object). *Given a* CHG $\langle \mathcal{C}, \prec_s, \prec_r \rangle$, *a sub-object* σ *is a pair* $[C, \langle X, Y_1, Y_2, \ldots, Y_n \rangle]$, *where*

1. $C, X, Y_1, Y_2, \ldots, Y_n \in \mathcal{C}$
2. $X \prec_r Y_1 \prec_r \ldots \prec_r Y_n$
3. $(C = X) \vee \exists (Z \in \mathcal{C})[C <_{sr} Z <_s X]$

C is the derived class to which σ belongs, and Y_n is the type of the sub-object. The path $X, Y_1, Y_2, \ldots, Y_n$ represents the path in the CHG through which class C inherits Y_n. For a repeated sub-object, $X = C$. For a shared sub-object, X is the least derived virtual base class that contains Y_n.

Example 4. Fig. 2(c) show the sub-objects of class B with replicated multiple inheritance. The sub-object $[B, \langle B, L, T \rangle]$ in class B corresponds to class T that is inherited transitively by class B through class L. The sub-object path $\langle B, L, T \rangle$ represents the corresponding path in the CHG. An instance of class B has two copies of T, one inherited from L and the other inherited from R. Therefore, there are two sub-objects $[B, \langle B, L, T \rangle]$ and $[B, \langle B, R, T \rangle]$ that correspond to class T. The sub-object path of T determines whether it is inherited from L or R.

Similarly, Fig. 3(c) shows the sub-objects of class B with shared inheritance. Note that an instance of class B has only one copy of class T. Therefore, there is only one sub-object that has an effective class type T, namely $[B, \langle T \rangle]$. The sub-object path $\langle T \rangle$ represents the fact that class B shared inherits T because the first class in the sub-object path is not B. ■

3 Clarifier

C++ provides convenient abstractions, such as constructors and destructors, that simplify the life of a software developer. However, such abstractions also introduce additional operations that are implicit in the control flow of the program. For example, the destructors of the objects allocated on the stack are implicitly invoked whenever the objects go out of scope. Examples of other such implicit operations are calls to constructors and overloaded operators, the `this` parameter in member functions, and implicit casts. The clarifier module exposes such implicit operations in the C++ program.

Example 5. Consider the following C++ program:

```
int cutLen(const string &s,size_t i,size_t n){
  const char *str = s.substr(i,n).c_str();
  return strlen(str);
}
```

The output of the clarifier is shown below:

```
int cutLen(const string &s,size_t i,size_t n){
  const string tmp; // Only declaration, no call to constructor.
  // Copy constructor call for ‘tmp’.
  tmp.string(s.substr(i,n));
  const char *str = tmp.c_str();
  // Destruction of temporary ‘tmp’
  tmp.~string();
  // Use of invalid pointer ‘str’
  return strlen(str);
}
```

In the output, the copy constructor for the temporary object that gets created at the call to `s.substr(...)` is made explicit. Similarly, the destructor for the temporary object is invoked when the temporary goes out of scope, which happens immediately after the initialization of `str`. Method call `tmp.c_str()` returns a pointer to an internal buffer, which is deallocated when the `tmp` object is destroyed. Therefore, `strlen` uses a deallocated string `str`, which can cause a segmentation fault.[1] The clarifier maintains the correct C++ semantics by preserving the order in which the constructors and destructors are invoked, and the order in which the initializations are performed.

The output of the clarifier is an intermediate representation called `CILpp` that is largely inspired by the CIL front-end [24] with relevant extensions for C++ specific features. The `CILpp` representation consists of a mixture of C and C++ constructs. Specifically, the inheritance-related constructs are still present. The inheritance-related features are eliminated by performing source-to-source transformations using the CHROME model as described next.

4 CHROME Model

Standard C++-to-C lowering algorithms use a *physical sub-object model* for representing objects of derived classes, which is similar to how compilers layout objects at runtime. In the physical sub-object model, the base classes are embedded into the derived

[1] This example is modeled on some bugs that our framework discovered in the *gold* project, which provides a faster linker as part of the GNU binutils package [2].

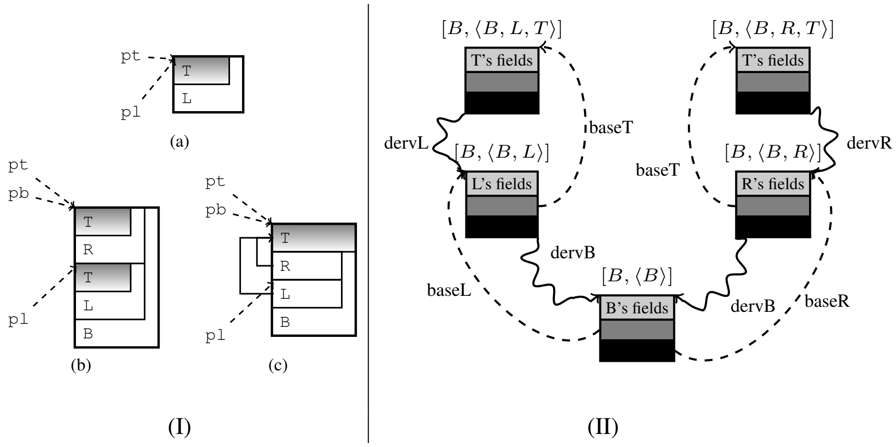


Fig. 4: (I) Object layout used by a standard C++-to-C lowering algorithm: (a) class L in Figs. 2 and 3, (b) class B with replicated inheritance in Fig. 2, and (c) class B with shared inheritance in Fig. 3. (II) CHROME object model for the class B in Fig. 2. (pt, pl, and pb are pointers.)

class objects. Fig. 4(I) shows the physical sub-object model for some classes that use replicated and shared multiple inheritance.

There are several problems with using the physical sub-object model for analysis. Because sub-objects are embedded inside the derived classes, a cast between a base and derived class pointer has to be modeled as an offset adjustment to the source pointer. For example, for the objects in Fig. 4(I), the cast statement "pl = (L*) pb", where pb is a pointer of type B*, requires moving pointer pb to the start of the corresponding sub-object of type L in the B-object as follows: "pl = ((char*)pb) + 8;".

In certain cases, the required offsets cannot be determined statically. For example, consider a cast from a pointer pl of type L* to a pointer of type T*. If pl points to a sub-object of type L as in Fig. 4(I)(a), no adjustment is necessary. On the other hand, if pl points to a sub-object of type L in an object of type B as in Fig. 4(I)(c), pointer pl has to be adjusted by 8 bytes. For such cases, the offsets are stored in the virtual-function pointer table and are consulted at runtime. Consequently, the code generated by a standard lowering algorithm includes a lookup of a virtual-function pointer table even for simple casts. As mentioned in Sect. 1, static analysis algorithms are not precise in the presence of such low-level pointer offset adjustments and arrays of pointers.

To avoid such problems, we propose the CHROME object model for representing the objects of a derived class. In the CHROME object model, an object is viewed as a collection of its sub-objects and no assumptions are made about the layout of the fields in each class. Whenever an object is created, the sub-objects that belong to the class are created independently, and are linked to each other via *additional* pointer fields.

Example 6. Consider the replicated inheritance hierarchy in Fig. 2. The CHROME object model representation for a B-object is shown in Fig. 4(II). The different sub-objects of the class are constructed separately and are connected to each other through additional pointer fields. For example, the dervL and baseT pointers connect the sub-objects $[B, \langle B, L\rangle]$ and $[B, \langle B, L, T\rangle]$. ■

These auxiliary object hierarchy edges are utilized by CHROME to walk the object when arbitration of inheritance related features is needed. As an example, consider casts, where, instead of computing pointer offset adjustments, we follow the additional pointers in the representation of the `B`-object. Next, we present the source-to-source transformations for various C++ constructs through examples.

Example 7 (Class Declarations). To facilitate the construction of the CHROME object model, we add the following fields to every class C in the CHG: (1) a `soid` field that is used to identify the sub-object that C represents, (2) a base pointer field and a derived pointer for every immediate base and derived class of C, respectively. For the program in Fig. 2, the classes are modified as follows (replicated multiple inheritance):

```
class T {int soid; L* dervL; R* dervR; int tf;}
class L {int soid; T* baseT; B* dervB; int lf;}
class R {int soid; T* baseT; B* dervB; int rf;}
class B {int soid; L* baseL; R* baseR; int bf;}
```

For the program in Fig. 3, classes `L`, and `R` are modified as shown above, and classes `T` and `B` are modified as follows (shared multiple inheritance):

```
class T {int soid; L* dervL; R* dervR; B* dervB; int tf;}
class B {int soid; L* baseL; R* baseR; T* baseT; int bf;}
```

Note that pointers `dervB` and `baseT` are added because `T` is a shared base class of `B`.

Example 8 (Object Construction). Consider the statement "`B* pb = new B()`", where `B` is the class from Fig. 2. As a first step, all the sub-objects of the given class `B` are allocated[2] and the `soid` and base pointer fields are initialized:

```
// Create sub-objects
// (see Fig. 2)
B* pb       = allocnew B();
L* pb_B_L   = allocnew L();
R* pb_B_R   = allocnew R();
T* pb_B_L_T = allocnew T();
T* pb_B_R_T = allocnew T();

// Set base pointer fields
pb->baseL        = p_B_L;
pb->baseR        = p_B_R;
pb->baseL->baseT = p_B_L_T;
pb->baseR->baseT = p_B_R_T;

// Set soid fields
pb->soid      = SOID([B, ⟨B⟩]);
pb_B_L->soid  = SOID([B, ⟨B, L⟩]);
pb_B_R->soid  = SOID([B, ⟨B, R⟩]);
pb_B_L_T->soid=SOID([B, ⟨B, L, T⟩]);
pb_B_R_T->soid=SOID([B, ⟨B, R, T⟩]);

// Invoke the constructor for B
pb->B();
```

In addition to the above steps, all the constructors are modified to initialize the derived pointer fields of every immediate base class and shared base class, and subsequently, invoke their constructors. For example, the constructor for `B` is modified as follows:

[2] `allocnew C()` allocates memory for an object of type `C` on the heap. It should be noted that we require that related calls to `allocnew()` either all succeed or the first one itself fails.

```
B::B(B* this) {
  this->baseL->dervB = this; this->baseL->L();
  this->baseR->dervB = this; this->baseR->R();
  ...
}
```

Similarly, the constructors of T, L, and R are also modified. The invocation of the constructors of L and R (which in turn would invoke the constructor of T) ensures that the sub-objects of B are initialized properly. ■

Note that while the number of sub-object *types* grows exponentially with the size of the class inheritance graph, the actual number of sub-object *instances* only grows linearly with the size of the class inheritance graph. Therefore, we have not seen the increase in the number of sub-objects affect the scalability of the analysis (see Sect. 5).

Example 9 (Cast and Field Accesses). Consider a cast statement "tgt =(T*)src", where tgt is of type T and src is of type S. First, all the sub-objects that src may legally point-to at runtime are determined. For every such sub-object σ, the access path ρ starting from src consisting of a sequence of derived and base pointer fields to reach the required T-sub-object is computed. Finally, a switch..case statement is generated with a case for every sub-object and access path pair $\langle\sigma, \rho\rangle$ that updates tgt.

Fig. 5 shows a few examples. Consider the translation for the cast statement "pb = (B*) pt". The sub-objects that pt may point-to at runtime are $[B, \langle B, L, T\rangle]$ and $[B, \langle B, R, T\rangle]$ (see Fig. 2(c)). The corresponding access paths are pt->dervL->dervB and pt->dervR->dervB, which are assigned to the target pointer pb in the respective cases. Note that if pt does not point to either of the two sub-objects, the target pointer pb is set to NULL, which mimics the semantics of a dynamic_cast.

The cast statement "pr = (R*)pt" in Fig. 5 demonstrates the case where the access path consists of derived pointer fields followed by base pointer fields. The translation is shown in Fig. 5 (after grouping common cases). Note that there is no case for $[L, \langle L, T\rangle]$ because (L $\not<_{sr}$ R), and therefore, $[L, \langle L, T\rangle]$ will not be in the set of sub-objects that pt may legally point to at runtime.

Consider a field access tgt = src->S::m. The statement is transformed in CHROME by considering it as tgt = ((S*)src)->S::m. The idea is to treat a field access as equivalent to the code sequence consisting of a cast of src to (S*) followed by the field access. Basically, the access path from src is generated and then the member is accessed. Fig. 5 shows the translation for z = pl->tf. (A similar strategy is used for member calls without dynamic dispatch.) ■

Example 10 (Virtual-function calls). For p->C::foo(p,...), where C::foo is a virtual function, all possible sub-objects that p could be pointing to at runtime is determined. Finally, a switch..case statement is generated with a case for each sub-object. The actual member function that would be invoked at run-time is called in each case of the switch statement. Note that, in each case, the access paths need to be adjusted accordingly for p. Consider the class hierarchy in Fig. 2. Suppose that class T defines a virtual function vfunc and class L overrides it. The virtual-function call

```
// Downcast: pb = (B*)pt
// (B* pb, T* pt)
switch(pt->soid) {
  case SOID([B, ⟨B, L, T⟩]):
    pb = pt->dervL->dervB;
    break;
  case SOID([B, ⟨B, R, T⟩]):
    pb = pt->dervR->dervB;
    break;

  default:
    pb = NULL;
}
```

```
// Cast: pr = (R*)pt
// (R* pr, T* pt)
switch(pt->soid) {
  case SOID([R, ⟨R, T⟩]):
  case SOID([B, ⟨B, R, T⟩]):
    pr = pt->dervR;
    break;
  case SOID([B, ⟨B, L, T⟩]):
    pr =
     pt->dervL->dervB->baseR;
    break;
  default:  pr = NULL;
}
```

```
// Upcast: pt = (T*)pl
// (T* pt, L* pl)
switch(pl->soid) {
  case SOID([B, ⟨B, L⟩]):
  case SOID([L, ⟨L⟩]):
    pt = pl->baseT;
    break;
 default: pt = NULL;
}
```

```
// Field access: z = pl->tf
// (int z, L* pl)
switch(pl->soid) {
  case SOID([B, ⟨B, L⟩]):
  case SOID([L, ⟨L⟩]):
    z = pl->baseT->tf;
    break;
  default: assert(false);
}
```

Fig. 5: CHROME translation for casts and field accesses with the class hierarchy in Fig. 2.

`pt->vfunc()` is translated as shown in Fig. 6. Note that if `pt` points to a sub-object inherited from L, `L::vfunc` is invoked. ■

Special Handling. It is easy to adapt the transformations described so far to the other constructs in the C++0x standard except for the following cases that require special attention. (1) For virtual-base classes, the CHROME transformations ensure that the body of the constructor (or destructor) of a virtual base class is only invoked once and it is always invoked from the constructor (or destructor) of the most derived object. (2) For virtual-function calls on an object that is being constructed, the CHROME lowering follows C++ semantics by treating the partially constructed object as though it is an object of the type to which the constructor belongs. (3) At object assignments, the CHROME lowering generates additional assignments that copies the sub-objects associated with the source into the sub-objects associated with the target. (4) Template classes and functions are instantiated. (5) Exceptional control-flow is made

```
switch(pt->soid) {
  case SOID([B, ⟨B, L, T⟩]):
  case SOID([L, ⟨L, T⟩]):
    pt->dervL->
      L::vfunc(pt->dervL);
    break;
  case SOID([B, ⟨B, R, T⟩]):
  case SOID([T, ⟨T⟩]):
    pt->T::vfunc(pt);
    break;
  default: assert(false);
}
```

Fig. 6: CHROME translation for a virtual-function call.

explicit for sound static analysis. We have presented an algorithm for transforming a C++ program with exceptions into a semantically equivalent C++ program without exceptions elsewhere [26]. The exception-elimination transformations can be performed in conjunction with the inheritance-elimination transformations described here.

Lowering without the `soid` *field.* For ease of presentation, the Chrome transformations presented so far used the `soid` field to determine valid paths in the sub-object graph. However, we can often omit the `soid` field because the auxiliary base class and derived class pointers themselves encode valid paths in the sub-object model; invalid paths are indicated by NULL values for the base class and derived class pointer fields. For example, the downcast in Fig. 5 can be translated without the `soid` field as follows:

```
// Downcast: pb = (B*)pt
// (B* pb, T* pt)
if (pt->dervR && pt->dervR->dervB) {
  pb = pt->dervR->dervB;
} else if (pt->dervL && pt->dervL->dervB) {
  pb = pt->dervL->dervB;
} else {
  pb = NULL;
}
```

The advantage of using the auxiliary pointers directly is that subsequent static analysis algorithms need not maintain the relationship between the value of the `soid` field and the auxiliary base and derived class pointers.

Correctness of the transformations. Here we provide the intuition as to why the transformations are semantics preserving. At object construction, it is easy to see that all the sub-objects are allocated and the fields are initialized by the chaining of constructor calls. After a cast statement, the target of a cast operation has to point to the relevant sub-object. To achieve this, compilers generate code that adjusts the pointer appropriately at runtime. The Chrome transformations mimic this behavior by generating access paths consisting of derived and base pointer fields. Because the derived and base pointer fields are set up to point to the correct sub-objects at object construction, the lowered code mimics the behavior of casts correctly. Similarly, at a virtual-function call, the appropriate member function is invoked based on the runtime type stored in the `soid` field, which is the expected behavior.

Because Chrome uses a different object layout, the memory behavior of a Chrome-lowered program is different from the original program. This is not an issue for the correctness of the transformations unless the program modifies the objects using low-level primitives like `memset`. However, programmers typically do not perform low-level operations like `memset` on objects (like they sometimes do in C) that use inheritance because it is highly compiler-dependent and can mess up the virtual-function pointer tables and other compiler-level data structures. Because we only change the objects that use inheritance, such low-level operations do not pose a problem for the semantic correctness of our transformations assuming compiler-independent code.

Table 1: **Characteristics of the C++ benchmarks.** LOC: the number of non-empty lines after preprocessing. **#class**: the number of classes from the standard libraries (lib) and from the application (app). **#mult**: number of classes with multiple inheritance in standard libraries (lib) and the actual application (app). **#VPTR**: number of accesses to the virtual-function pointer table. **#FPTR**: number of function-pointer calls. **#T**: time taken for CHROME-lowering in seconds.

	C++ program						Lowered C Program				
							COMPILER			CHROME	
			#class		#mult						
	LOC	#T	lib	app	lib	app	LOC	#VPTR	#FPTR	LOC	#FPTR
coldet (1.2)	5.1K	1.7s	32	61	0	4	7.0K	48	31	14.4K	0
mailutils (2.1)	8.3K	2.1s	14	106	0	0	8.7K	2	0	17.1K	0
tinyxml (2.5.3)	4.9K	2.0s	0	59	0	0	12.5K	110	79	21.6K	0
id3lib (3.8.3)	14.5K	8.0s	75	106	6	0	35.7K	632	499	77.7K	0
cppcheck (1.4.3)	30.9K	30s	148	104	7	5	99.9K	217	71	165.0K	0

5 Implementation and Experiments

We have implemented the ideas described in the paper in an in-house extension of CIL [24] called `CILpp`. The C++ front-end for `CILpp` is based on `EDG` [15], and therefore, handles all aspects of the C++0x standard. For the experiments, we translated the given C++ program into an equivalent C program using (1) a standard compiler-based lowering mechanism, and (2) the lowering mechanism based on the CHROME object model. Henceforth, we refer to the C program obtained from compiler-based lowering as COMPILER-lowered C program, and the one obtained from CHROME-based lowering as CHROME-lowered C program.

Tab. 1 shows the characteristics of the open source benchmarks used for our experiments. The open source library `coldet` (v1.2) implements collision detection algorithms and is often used in game programming. GNU `mailutils` (v1.2) is a collection of mail utilities, servers, and clients. `TinyXML` (v2.5.3) is a light-weight XML parser which is widely used in open source and commercial products. The open source library `id3lib` (v3.8.3) is used for reading, writing, and manipulating ID3v1 and ID3v2 tags, which is the metadata format for MP3 audio files. `cppcheck` (v1.4.3) is a tool for static C/C++ code analysis[3] that uses a library of problematic code patterns to detect common errors. For the experiments, the sources relevant to the project were merged into a single C++ file and preprocessed. The preprocessed file was lowered using the compiler-based and CHROME-based lowering mechanisms.

5.1 Complexity of the lowered C programs

The column labeled "LOC" in Tab. 1 refers to the number of non-empty lines of code in the merged file after preprocessing. The COMPILER-lowered C program is up to 3

[3] We also ran `cppcheck` on our set of benchmarks, but it did not find any of bugs reported in Sect. 5.2. The patterns used by `cppcheck` are not sufficient enough to find bugs that deep static analyzers are capable of detecting.

times larger than the original C++. For the mailutils example, which has no virtual-function calls, the size of the COMPILER-lowered program is roughly the same as the original C++ program. This suggests that the extra statements generated by compiler-based lowering mostly relate to the setup and access of virtual-function pointer tables. The CHROME-lowered C program is roughly *three to five* times the size of the original C++ program, and is roughly *twice* that of the COMPILER-lowered C program. The difference in the sizes between COMPILER-based and CHROME-based lowering is mostly due to `switch..case` statements generated by the CHROME transformations. Even though the COMPILER-lowered programs are smaller than the CHROME-lowered programs, COMPILER-lowered programs contain operations that are hard for a static analyzer to reason about, such as calls via function pointers and accesses to virtual-function pointer tables. The column labeled “#FPTR” shows the number of calls through function pointers, and the column labeled “#VPTR” shows the number of accesses to virtual-function pointer tables. The function pointer calls in COMPILER-lowered program correspond to the virtual-function calls in the original C++ program. The CHROME-lowered programs do not have calls via function pointers because CHROME-based transformations do not use function pointers for virtual-function calls. Note that the number of accesses to a virtual-function pointer table is generally more than the number of calls via function pointers, which indicates that virtual-function pointer tables are also used for purposes other than dispatching virtual-function calls.

5.2 Effectiveness of Lowering for Software Verification

Table 2: Results of memory leak and pointer validity checker using F-SOFT on the lowered programs. **#N**: the number of NULL-pointer dereferences. **#M**: the number of memory leaks. The number in parenthesis shows the number of real bugs. The time limit was set to 20 minutes for each function.

	COMPILER		CHROME	
	#N	#M	#N	#M
coldet	0	0	5(2)	0
mailutils	0	0	0	0
tinyxml	3(0)	0	0	2(0)
id3lib	4(3)	1(1)	6(5)	6(1)
cppcheck	1(1)	0	3(2)	1(0)

For this experiment, we analyzed the COMPILER-lowered and CHROME-lowered programs using F-SOFT [18]. F-SOFT is a tool for finding bugs in C programs, and uses a combination of abstract interpretation and model checking to find common programming mistakes, such as NULL-pointer dereferences, memory leaks, buffer overruns, and so on. Given a C program, F-SOFT systematically instruments the program in such a way that an assertion is triggered whenever a safety property is violated. For example, at every dereference of a pointer, the program is instrumented to trigger an assertion if the pointer is NULL. An abstract interpreter [12] is used as a proof engine in F-SOFT. The abstract interpreter computes invariants that can be used to prove that certain assertions can never be reached. Our abstract interpreter is inter-procedural, flow and context sensitive. It is built in a domain-independent and extensible fashion, allowing for various abstract domains such as constants, intervals [11], octagons [22], symbolic ranges [31] and polyhedra [14]. These domains are applied in increasing order of complexity. After each analysis is run, the assertions that are proved to be unreachable are removed and

```
const uchar* ID3_FieldImpl::GetRawBinary() const
{
  const uchar* data = NULL;
  if (this->GetType() == ID3FTY_BINARY) {
    data = _binary.data();
  }
  return data;
}
void ID3_FieldImpl::RenderBinary(ID3_Writer& writer)
{
  writer.writeChars(this->GetRawBinary(),
                    this->Size());
}
```

Fig. 7: NULL pointer dereference in `id3lib`.

the program is simplified by constant propagation and slicing. After abstract interpretation, the remaining properties are checked by a SAT-based bounded model checker. If the model checker finds any violations, they are reported to the user along with a witness trace. For the sake of usability, F-Soft does not report sound analysis results, and uses heuristics to identify patterns that commonly result in false warnings and eliminates them.

Tab. 2 summarizes the results of a pointer-validity checker and a memory-leak checker in F-Soft, where the number of witnesses reported by the model checker is presented along with the number of real bugs reported in parenthesis. The time limit was set to 20 minutes for each function. It should be noted that the lowering techniques produce entirely different (but semantically equivalent) programs. Therefore, the number of properties in the Chrome-lowered program is different from the number of properties in the Compiler-lowered program. In summary, F-Soft found a total of ten real bugs, of which five were found only when F-Soft analyzed the Chrome-lowered program. Note, that all of these bugs were previously unknown. Finally, we note that all witnesses found using the Compiler-lowered were also found by F-Soft when analyzing Chrome-lowered program.

In the following, we highlight a few bugs that were found using the Chrome-lowered C program. For the Chrome-lowered programs, F-Soft reported a total of 14 NULL-pointer dereference witnesses of which 9 were found to be real bugs.

NULL-pointer dereference in virtual-function calls. Consider the code snippet from `id3lib` shown in Fig. 7. It is possible to dereference a NULL pointer as follows: the body of `writeChars` method (not shown) assumes that the first argument to `writeChars` is never NULL, but `GetRawBinary` method may return NULL. The class hierarchy in `id3lib` is not trivial: `ID3_FieldImpl` is derived from `ID_Field` and `ID3_Writer` is a base class for 9 derived classes of which 7 are immediate derived classes. Further, all the methods invocations in Fig. 7 are virtual. When analyzing the Chrome-lowered C program, F-Soft presents a witness that invokes `RenderBinary` with an object of type `ID3_FieldImpl` for the `this` pointer and an object of type `UnsyncedWriter` for the reference parameter `writer`, followed by a call to the method `UnsyncedWriter::writeChars`, where the NULL

Table 3: Effectiveness of Chrome in CBMC. C++ features are as follows: inheritance (Inh), multiple inheritance (Mi), dynamic cast (Dc), and virtual functions (Vf). Results are as follows: false positives (FP), false negative (FN), front-end failure (FF), and OK (√).

	C++ Features				CBMC	CBMC		F-Soft	
	Inh	Mi	Dc	Vf	`goto-cc`	Compiler	Chrome	Compiler	Chrome
P1	√				FP	√	√	FN	√
P2	√	√			FF	√	√	FN	√
P3	√		√		FN	FN	√	FN	√
P4	√			√	√	FN	√	FN	√
P5	√		√	√	FN	FN	√	FN	√
P6	√	√	√	√	FF	FN	√	FN	√

pointer dereference occurs. This bug was not found when the Compiler-lowered code is analyzed by F-Soft because of the use of a virtual-function pointer table at virtual-function calls.

Interprocedural NULL pointer dereferences. The analysis also discovered four more scenarios where a NULL pointer dereference may occur in `id3lib`. These are related to methods that return NULL pointers instead of valid C-strings under certain error conditions. However, at certain call-sites to these methods, the returned C-string is passed on to string manipulation functions such as `strcmp` without checking if the returned string is NULL, thereby potentially causing segmentation faults. In addition, F-Soft found four bugs of a similar kind in `coldet` and `cppcheck`.

Finally, in the `TiXmlComment::Parse` method of the `TinyXML` project, F-Soft reported a NULL-pointer dereference. The NULL pointer dereference occurs when the input string to `TiXmlComment::Parse` is empty. However, from further investigation , it seems unlikely that this function will be called using an empty string.

Memory Leaks. When analyzing the Chrome-lowered program for memory leaks, it reported 9 warnings, of which 1 was a real memory leak. The memory leak happens in a method named `convert_i` in file `utils.cc` of `id3lib`. F-Soft reported this leak for the Compiler-lowered program also.

5.3 Applicability in other verification tools

In addition to using F-Soft, we also wanted to investigate the applicability of our lowering in other verification tools. For this experiment, we created a collection of microbenchmarks [23] that exercises various aspects of C++. Each program has two assertions of which one always fails and the other always succeeds at runtime. For each benchmark, we generated the Compiler-lowered and Chrome-lowered programs and analyzed them using the CBMC verfication tool [8]. The CBMC suite uses the `goto-cc` C++ front-end to generate an intermediate representation for analysis.

Tab. 3 shows the results of our experiments. The column labeled `goto-cc` shows the results of running CBMC directly on the C++ program. A false negative refers to the case where the tool does not report the failing assertion. A false positive refers to the case where the tool reports an error for the succeeding assertion. As the table shows CBMC with `goto-cc` does not perform well when complex C++ features such as multiple inheritance and dynamic casts are used. It generates a false positive or a false negative in many cases. The results highlight how difficult and error prone it is to encode C++ semantics in program analyzers. When CBMC is used to analyze the COMPILER-lowered program, it generates false negatives in many cases.

On the other hand, when CBMC is used to analyze the CHROME-lowered program, it does not generate any false reports. (F-SOFT also performs well when a CHROME-lowered program is analyzed.) This points to the effectiveness of using the CHROME transformations even in other verification tools.

6 Related Work

The sub-object formalism presented by Rossie and Friedman forms the basis of our CHROME object model transformations [30]. Ramalingam and Srinivasan present a member lookup algorithm for C++ [28] which operates directly on the class hierarchy graph (CHG) instead of the sub-object graph, which may be exponential in the size of the CHG. The issue of member lookup is orthogonal, but complementary, to the problem of translating a C++ program with inheritance into a semantically equivalent program without inheritance. Based on the Rossie-Friedman model, a number of class hierarchy transformations have also been proposed which preserve the program semantics, e.g., by slicing class hierarchies in libraries according to their clients for producing optimized code [38].

Various approaches formalize the object layout in C++ to study space overhead and performance aspects of object layouts [35, 36], and perform formal verification of object layouts [5, 29, 39]. These formalisms are geared towards devising memory efficient layouts and the correctness of the layout algorithms. In contrast, the goal of CHROME is to embed the object model into the program to make it more amenable to static analysis. However, such formalisms are complementary to our approach and may be used to establish the correctness of CHROME transformations.

Another work that closely relates to this paper is the LLVM compiler framework [19]. The LLVM framework translates a given C++ program into a low-level three address code. Unlike our lowering algorithm, LLVM algorithm uses virtual function tables and runtime libraries during lowering, and therefore, produces code that is not very amenable for precise program analysis.

The ROSE compiler front-end [27] and Clang static analyzer [6] are other popular front-ends that generate an Intermediate Representation (IR) for C++ programs. These front-ends support all C++ language features. However, the IR produced by these front-ends still contain complex C++ features such as inheritance and casts. Therefore, every analysis that uses their IR has to deal with inheritance and casts. On the other hand, the source-to-source transformations presented in this paper eliminate complex C++ features, thereby making the implementation of subsequent analysis easier. The techniques presented here may be used to simplify the IRs generated by ROSE and Clang.

A combination of the notions of delegation [21] (instantiating additional object fields and forwarding method calls to them) and interfaces has been used [37] to simulate multiple inheritance in languages. However, these methods do not delve into the complexities of C++ object model, e.g., handling shared and replicated inheritance, casting, etc. The CHROME transformations, in contrast, handle these features precisely.

Chen [5] proposed a typed intermediate language (IL) and a method to compile multiple inheritance into the language. The IL, however, is quite mathematical in nature and dedicated analyses must be designed for the IL. In contrast, our target language is (a subset of) C++ itself, and hence, the target program is immediately amenable to conventional static analysis.

Finally, there is a vast literature on analysis (static or dynamic) of object-oriented programs, mostly focused on Java programs [20]. In particular, Chandra et al. proposed directed call graph construction [4] for Java programs to handle the explosion in the number of potential virtual method calls, by interleaving call graph construction with backward symbolic analysis.

7 Conclusions

In this paper, we presented an algorithm to translate a C++ program with inheritance into a C++ program without inheritance using a representation of sub-objects called CHROME. We also showed the effectiveness of the CHROME lowering on program analysis applications such as software model checking. The C program obtained using the CHROME-based transformations enabled better results than the C program obtained from a standard compiler-based lowering algorithm. We found a total of ten previously unknown bugs, of which *five* were only found due to the precise modeling of objects using CHROME. The results are quite encouraging and validates that our CHROME-lowered code is better suited for program analysis.

References

1. Apache. Xerces project. (http://xerces.apache.org/).
2. G. Balakrishan, N. Maeda, S. Sankaranarayanan, F. Ivančić, A. Gupta, and R. Pothengil. Modeling and analyzing the interaction of C and C++ strings. In *Int. Conf. on Formal Verif. of Object-Oriented Software*, 2011.
3. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
4. S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A powerful approach to weakest preconditions. In *PLDI*, pages 363–374, 2009.
5. J. Chen. A typed intermediate language for compiling multiple inheritance. In *POPL*, 2007.
6. Clang static analyzer. http://clang-analyzer.llvm.org.
7. C++ support for Clang. http://clang-analyzer.llvm.org/dev_cxx.html, (accessed Jan 6, 2012).
8. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *TACAS*, pages 168–176, 2004.
9. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
10. Comeau C++ compiler. www.comeaucomputing.com.
11. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd. Int. Symp on Programming*, Paris, Apr. 1976.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.
13. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, pages 21–30, 2005.

14. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, pages 84–96, 1978.
15. C++ frontend. Edison Design Group, NJ.
16. K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4):366–381, 2000.
17. D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE*, pages 9–14, 2007.
18. F. Ivančić, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-Soft. In *IEEE International Conference on Computer Design*, pages 297 – 308, Oct. 2005.
19. C. Lattner. LLVM: A compilation framework for lifelong program analysis and transformation. In *Int. Symp. on Code Generation and Optimization*, 2004.
20. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.
21. H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*, pages 214–223, 1986.
22. A. Miné. The octagon abstract domain. In *Working Conf. on Rev. Eng.*, 2001.
23. NECLA verification benchmarks. http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php.
24. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.
25. Nokia. Qt: a cross-platform application and UI framework. (http://qt.nokia.com/).
26. P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić, and A. Gupta. Interprocedural exception analysis for C++. In *European. Conf. on Object-oriented Programming*, 2011.
27. D. J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.
28. G. Ramalingam and H. Srinivasan. A member lookup algorithm for C++. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 18–30, New York, NY, USA, 1997. ACM.
29. T. Ramananandro, G. D. Reis, and X. Leroy. Formal verification of object layout for C++ multiple inheritance. In *POPL*, 2011.
30. J. G. Rossie, Jr. and D. P. Friedman. An algebraic semantics of subobjects. In *OOPSLA*, pages 187–199, New York, NY, USA, 1995. ACM.
31. S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis using symbolic ranges. In *SAS*, volume 4634 of *Lec. Notes in Comp. Sci.*, pages 366–383, 2007.
32. C. standards commitee. Working draft, standard for C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, (accessed Jan 6, 2012).
33. B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, 1989.
34. B. Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *Proc. of History of Programming Languages III*, 2007.
35. P. F. Sweeney and M. Burke. Quantifying and evaluating the space overhead for alternative C++ memory layouts. *Softw. Pract. Exper.*, 33(7):595–636, 2003.
36. P. F. Sweeney and J. Y. Gil. Space and time-efficient memory layout for multiple inheritance. In *OOPSLA*, New York, NY, USA, 1999. ACM.
37. K. Thirunarayan, G. Kniesel, and H. Hampapuram. Simulating multiple inheritance and generics in Java. *Comp. Lang.*, 25(4):189–210, 1999.
38. F. Tip and P. F. Sweeney. Class hierarchy specialization. *Acta Inf.*, 36(12):927–982, 2000.
39. D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA*, pages 345–362. ACM Press, 2006.
40. Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *Trans. on Prog. Lang. and Syst.*, 29(3), 2007.