NASA/CP—2009–215403

Proceedings of the Third International Workshop on Proof-Carrying Code and Software Certification

Edited by

Ewen Denney
Thomas Jensen

Ames Research Center, Moffett Field, California

October 2009

NASA STI Program ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing help desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

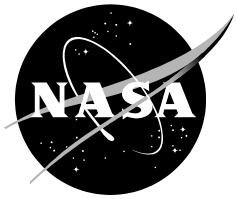
- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at 443-757-5803

- Phone the NASA STI Help Desk at 443-757-5802

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CP—2009–215403



Proceedings of the Third International Workshop on Proof-Carrying Code and Software Certification

Edited by

Ewen Denney
Thomas Jensen

Ames Research Center, Moffett Field, California

Prepared for
The Third International Workshop on
Proof-Carrying Code and Software Certification
Sponsored by IEEE
Los Angeles, CA, August 15, 2009

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, CA 94035-1000

October 2009

Available from:
NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Preface

This NASA conference publication contains the proceedings of the Third International Workshop on Proof-Carrying Code and Software Certification, held in Los Angeles, CA, USA, on August 15, 2009 and co-located with the Logic in Computer Science (LICS) conference.

Software certification demonstrates the reliability, safety, or security of software systems in such a way that it can be checked by an independent authority with minimal trust in the techniques and tools used in the certification process itself. It can build on existing validation and verification (V&V) techniques but introduces the notion of explicit software certificates, which contain all the information necessary for an independent assessment of the demonstrated properties. One such example is proof-carrying code (PCC) which is an important and distinctive approach to enhancing trust in programs. It provides a practical framework for independent assurance of program behaviour; especially where source code is not available, or the code author and user are unknown to each other.

The workshop addressed the theoretical foundations of logic-based software certification as well as practical examples and work on alternative application domains. Here "certificate" is construed broadly, to include not just mathematical derivations and proofs but also safety and assurance cases, or any formal evidence that supports the semantic analysis of programs: that is, evidence about an intrinsic property of code and its behaviour that can be independently checked by any user, intermediary, or third party. These guarantees mean that software certificates raise trust in the code itself, distinct from and complementary to any existing trust in the creator of the code, the process used to produce it, or its distributor.

In addition to the contributed talks, the workshop also featured two invited talks, by Kelly Hayhurst, NASA Langley, and Andrew Appel, Princeton University.

We would like to thank the program committee members for their work. Thanks also go to Allen Dutra for his help with the PCC 2009 website and other organization, and to the EasyChair organizers for the use of their website and style files.

The PCC 2009 website can be found at `http://ti.arc.nasa.gov/event/pcc09/`.

August 2009

Ewen Denney
Thomas Jensen

PCC 2009 Chairs

Workshop Organization

General Chairs

Ewen Denney, SGT/NASA Ames
Thomas Jensen, CNRS/IRISA

Program Committee

David Aspinall, University of Edinburgh
Gilles Barthe, IMDEA Software
Ewen Denney, SGT/NASA Ames
Bernd Fischer, University of Southampton
Sofia Guerra, Adelard
Kelly Hayhurst, NASA Langley
Thomas Jensen, IRISA/CNRS
David Pichardie, INRIA
Germán Puebla, Technical University of Madrid
Ian Stark, University of Edinburgh

External Reviewers

Nurlida Basir, University of Southampton

Table of Contents

Mending the Gap, An effort to aid the transfer of formal methods technology

Kelly Hayhurst
NASA Langley Research Center
Hampton, VA

Abstract

Formal methods can be applied to many of the development and verification activities required for civil avionics software. RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, gives a brief description of using formal methods as an alternate method of compliance with the objectives of that standard. Despite this, the avionics industry at large has been hesitant to adopt formal methods, with few developers have actually used formal methods for certification credit. Why is this so, given the volume of evidence of the benefits of formal methods? This presentation will explore some of the challenges to using formal methods in a certification context and describe the effort by the Formal Methods Subgroup of RTCA SC-205/EUROCAE WG-71 to develop guidance to make the use of formal methods a recognized approach.

E. Denney, T. Jensen (eds.); The 3rd International Workshop on Proof Carrying Code and Software Certification, pp. 1-1

Proof-Carrying Code with Correct Compilers

Andrew W. Appel
Princeton University
Princeton, NJ

Abstract

In the late 1990s, proof-carrying code was able to produce machine-checkable safety proofs for machine-language programs even though (1) it was impractical to prove correctness properties of source programs and (2) it was impractical to prove correctness of compilers. But now it is practical to prove some correctness properties of source programs, and it is practical to prove correctness of optimizing compilers. We can produce more expressive proof-carrying code, that can guarantee correctness properties for machine code and not just safety. We will construct program logics for source languages, prove them sound w.r.t. the operational semantics of the input language for a proved-correct compiler, and then use these logics as a basis for proving the soundness of static analyses.

E. Denney, T. Jensen (eds.); The 3rd International Workshop on Proof Carrying Code and Software Certification, pp. 2-2

Efficient Type Representation in TAL

Juan Chen
Microsoft Research
juanchen@microsoft.com

Abstract

Certifying compilers generate proofs for low-level code that guarantee safety properties of the code. Type information is an essential part of safety proofs. But the size of type information remains a concern for certifying compilers in practice. This paper demonstrates type representation techniques in a large-scale compiler that achieves both concise type information and efficient type checking. In our 200,000-line certifying compiler, the size of type information is about 36% of the size of *pure code and data* for our benchmarks, the best result to the best of our knowledge. The type checking time is about 2% of the compilation time.

1 Introduction

Proof-Carrying Code (PCC) [9] and Typed Assembly Language (TAL) [8] use certifying compilers to generate proofs for low-level code that guarantee safety properties of the code[1]. Type information is an essential part of safety proofs and is used to describe loop invariants, preconditions of functions, etc. The size of type information, though, remains one of the main concerns for certifying compilers in practice. With naive representations, the size of type information can explode. Even with techniques that greatly improve sharing of type representations, such as sharing common subterms among types, type information may still be much larger than code. This is mainly because type checking needs elaborate type information to describe machine states, including the register bank, the heap, and the stack. Large type size is undesirable in practice, especially for large programs.

This paper demonstrates type representation techniques in a large-scale compiler that achieves both concise type information and efficient type checking. In our 200,000-line object-oriented certifying compiler, the size of type information is about 36% of the size of *pure code and data* for our benchmarks, the best result to the best of our knowledge. The type checking time is about 2% of the compilation time. Note here that we compare type information size against code and data size, *not* object file size. Object files contain relocation and symbol information and may be much larger than pure code and data.

Our compiler uses simple yet effective type representation techniques—some shared with existing certifying compilers (mostly for functional languages) and others specialized for object-oriented languages: (1) it shares common subterms of types (CSE on types); (2) it uses function types to encode more elaborate code pointer types; (3) it removes redundant or unnecessary metadata (information about classes and interfaces, including fields, methods, and inheritance); (4) it removes type information for components of large data in the data sections of object files; (5) it uses concise integer encodings.

The contribution of this paper is that it shows that large-scale certifying compilers can have both concise type representation and efficient type checking for TAL. The paper also shows that type representation in object-oriented certifying compilers needs special care with metadata and data.

2 Background

We first give an overview of the compiler, the type checker, and the type system of the target language.

E. Denney, T. Jensen (eds.): The 3rd International Workshop on Proof Carrying Code and Software Certification, volume 0, issue: 0, pp. 3-12

[1] We refer readers to Pierce's book for thorough introductions to PCC and TAL [12].

The Compiler. Our certifying compiler (called Bartok) compiles Microsoft Common Intermediate Language (CIL) to standalone typed x86 executables [2]. It has about 200,000 lines of code (mostly in C#) and more than 40 optimizations. Bartok is about an order of magnitude larger than previously published certifying compilers. It is used by about 30 developers on a daily basis. Performance of Bartok's generated code is comparable to that under the Microsoft Common Language Runtime (CLR). According to the benchmarks tested, programs compiled by Bartok are 0.94 to 4.13 times faster than the CLR versions, with a geometric mean of 1.66.

Bartok compiles CIL through several typed intermediate languages and generates typed x86. Unlike most existing certifying compilers which mingle type information and code, Bartok separates them. It writes type information in a special section in the object file. The sections for code and data use the standard Common Object File Format (COFF) [5] without any change. A standard linker links the object file with libraries and generates normal x86 executables, the same way it links untyped x86 object files. The linker discards the type information section.

Separating types from code allows us to keep the standard code format, but it needs to record additional information to connect the type with the code, for example, where a type is used. We chose this approach so that Bartok-compiled code can link with the libraries, which are not compiled by the certifying compiler at the moment.

The Type Checker. We would like to make the type checker simple and efficient because the type checker is in the trusted computing base. One design choice is that the type checker checks each basic block in the program only once. The TALx86 compiler [7]—a certifying compiler from a C-like language to x86—has a threshold that decides how many times a block can be checked (4 is the threshold in [6]).

The checker checks one function at a time. A function consists of a sequence of basic blocks. The checker performs a breath-first scan of the control flow graph, starting from the root block.

Checking a basic block requires the block's precondition, i.e., the state of the register bank and the control stack at the beginning of the block. The precondition either comes from type annotations for the block or can be computed by the type checker.

Given the precondition, the type checker checks the instructions in a basic block. Checking an instruction may need additional type information. For example, if an instruction refers to static data, the type checker needs type information for the data. The additional type information for instructions comes from annotations in the type section.

The type checker uses the postcondition of a basic block to compute the preconditions of the successors, if the successors do not have their preconditions already. A block at merge points must have an annotation for its precondition, unless all its predecessors have the same postcondition. This approach, together with the breadth-first scan, guarantees that the checker checks each block only once.

Exception header blocks (for exception handlers) always have annotations for their preconditions because there is no explicit control flow from the root block to the exception header blocks.

The Target Type System. The type system of the target language uses main ideas of LIL_C [3] to represent classes and objects. LIL_C is a low-level typed intermediate language for compiling object-oriented languages with classes. It is lower-level than bytecode and CIL because it describes implementation of virtual method invocation and type cast instead of treating them as primitives.

LIL_C differs from prior class and object encodings in that it preserves object-oriented notions such as class names and name-based subclassing, whereas prior encodings compiled these notions away.

LIL_C uses an "exact" notion of classes. A class name in LIL_C represents only objects of exactly that class, not including objects of subclasses. Each class C has a corresponding record type $R(C)$ that describes the object layout of C, including the vtable and the fields.

Objects can be coerced to or from records with appropriate record types, without runtime overhead. To create an object, we create a record and coerce it to an object. To fetch a field or invoke a method, we coerce the object to a record and then fetch the field or the method pointer from the record.

To represent an object of C or C's subclasses, LIL_C uses an existential type $\exists\alpha \ll C.\ \alpha$, read as "there exists α where α is a subclass of C, and the object has type α". The type variable α represents the object's runtime type and the notation $\ll$ means subclassing. The type variable α has a subclassing bound C, which means that the runtime type of the object is a subclass of C. The subclassing bounds can only be class names or type variables that will be instantiated with class names. Subclassing-bounded quantification and the separation between name-based subclassing and structure-based subtyping make the type system decidable and expressive.

A source class name C is then translated to an LIL_C type $\exists\alpha \ll C.\ \alpha$. If C is a subclass of class B, then $\exists\alpha \ll C.\ \alpha$ is a subtype of $\exists\alpha \ll B.\ \alpha$, which expresses inheritance—objects of C or C's subclasses can be used as B objects.

To represent the precondition of a basic block, the target language uses a code pointer type to describe the types of registers and stack locations at the beginning of the block. The code pointer type is elaborate: it specifies the type for each live register and for each stack location in the current stack frame, including the return address.

3 Type Representation

Now we explain the main type representation techniques in Bartok. The first technique shares common subterms among types, a common practice. Traditional sharing is important but not enough. Our compiler combines it with several other techniques: the second technique optimizes representation of code pointer types; the third one compresses representation of metadata information; the fourth reduces type information for data; and the last one improves integer encoding. The third and the fourth ones are specific to object-oriented languages, reducing type size by one third in our measured benchmarks.

One may think that we make the type checker simple but the type decoder complicated because of these type representation techniques. We consider this a good tradeoff because the type decoder is not in the trusted computing base. The result of the type decoder is checked and thus not trusted, just like untrusted type annotations. The type checking fails if the type decoder generates type information incompatible with the code.

3.1 Sharing Common Subterms

Sharing common subterms of types is a well known technique many compilers use ([6, 14]). It is similar to the common sub-expression elimination (CSE) optimization for terms, where two types that have a common subterm share one copy of the subterm, instead of duplicating it in both types.

When writing type information, Bartok first collects types that should be written into the object files. It goes through the whole program, including data, external labels, functions, basic blocks, and instructions, and collects each type it encounters, and records the type in an array. The compiler keeps track of which type it has seen. If it sees a type that has appeared already, nothing needs to be done. Otherwise, it marks the type as seen and continues to collect the components (subterms) of the types. Collecting component types makes it possible to achieve finer-grained sharing between types.

The compiler also tracks where in the program these types are used—the relative location in the code section of the corresponding basic blocks, instructions, etc. Such information is also written into the object file to connect types with the code those types annotate.

Bartok then writes the collected types into the object file. It writes the type array first, which contains all types the program uses. Each type starts with a byte indicating whether the type is a primitive type (e.g. int32) or a type constructor (e.g. array). If the first byte is a type constructor, the following bytes are encodings of the components. Interpretation of the following types depends on the type constructor. For example, if the type constructor indicates this is an array type, the following bytes encode the rank and the element type.

The encoding of a component type is its index in the type array. The index is valid because all component types are collected in the type array as well. The indirection achieves sharing in the object file: two occurrences of the same type share the same encoding.

After the compiler finishes writing the type array, it writes the types for block preconditions, instructions, data, etc. All types are again encoded as their indices in the type array.

When the type decoder reads back from object files the type information, it first decodes the type array and rebuilds the types. Because of the type index representation, the decoded types are hash-consed automatically. This also makes the type checker efficient because type equality tests are simply reference equality tests. Two structurally equivalent types share the same reference.

3.2 Code Pointer Types

The target language uses code pointer types to represent preconditions. As Grossman and Morrisett pointed out in [6], the size of preconditions is the most important factor of type size. Code pointer types are pervasive: they can take up to 20% of types used by programs. Furthermore, code pointer types are more complex and larger than other types because they record machine states. Optimizing code pointer type representation can have significant impact on type size.

There are two uses of code pointer types in Bartok: for method types and for preconditions of basic blocks. The source language and the high-level intermediate representations use function types for methods. The target language uses code pointer types to express explicit control stacks. Function types specify the types for the parameters and the return value and have no reference to the control stack. To check an x86 call instruction, the type checker needs explicit description of the current stack to check if the stack satisfies the requirement of the callee.

One observation is that if we know the function type and the calling convention of a method, we can compute the corresponding code pointer type. Therefore we can use the function type and the calling convention as a more concise representation of the code pointer type, which saves space dramatically. There are only a few calling conventions Bartok uses, so encoding calling conventions is easy. This technique reduces the number of code pointer types by 90% and the total number of types by 76% in our benchmarks.

The type decoder transforms function types to code pointer types according to their calling conventions. The type checker does not see any function types at all, therefore does not need any special handling for function types.

This approach has the drawback that the decoder has to hard-wire the calling conventions the compiler uses. But note here that the compiler still has the flexibility to use more than one calling convention or choose from various calling conventions. We require only that the decoder be aware of all the calling conventions the compile may use. We think the benefits of using function types outweigh the drawback.

We have addressed the code pointer types for methods, but preconditions for basic blocks still need code pointer types and they may not have corresponding function types. Bartok reduces the number of preconditions by omitting preconditions of basic blocks with only one predecessor or with multiple predecessors where the post conditions of all predecessors agree. Such preconditions can be reconstructed easily.

For the remaining code pointer types, Bartok optimizes common patterns, for example, sharing types of callee-save registers in preconditions. Bartok uses the standard approach to handle callee-save registers: at the entry point of a method, each callee-save register is assigned a type variable. The return address requires that the callee-save register have the same type variable to guarantee that the value of the register has been restored when the method returns. Every method has type variables for these callee-save registers. Those type variables are carried into preconditions of each basic block. Furthermore, if a callee-save register is not assigned a new value, its type remains the original type variable. The compiler shares the type variables for those registers and uses bits in preconditions to indicate if a callee-save register still has not changed its value.

3.3 Class Metadata

Object-oriented programming languages use metadata extensively to represent essential information about classes and interfaces, such as fields, methods, and inheritance. Object-oriented programs are organized by classes. Classes may contain many fields and methods and thus require large metadata. Also it is common for object-oriented programs to refer to classes defined in libraries or in other compilation units. Metadata for those externally defined classes is needed as well.

Checking core object-oriented features such as field access and virtual method call relies on metadata. To type check an instruction that fetches a field from an object, the checker needs to know the type of the field. Similarly, to type check a virtual method call instruction, the checker needs to know the type of the method in order to check whether the arguments have the desired types. The information about field types and method types is represented as metadata for the class.

At assembly language level, all field accesses and method accesses are lowered to memory accesses, with field/method names lowered to integer offsets from the beginning of the object reference/the vtable. Metadata in the target language records mapping from offsets of fields and methods to their types.

Metadata in the target language also includes inheritance information, for example, the superclass and superinterfaces of a class. This is important for the type checker to decide if a class/interface is a subclass of another class/interface.

Bartok includes in the metadata of a class the following information: field offsets and types, virtual method offsets (in vtables) and types, the superclass, and the superinterfaces. For interfaces, metadata contains superinterfaces and interface method offsets (in interface tables) and types.

Bartok has the following techniques to reduce metadata information in object files. First, it shares metadata information between superclasses and subclasses. A subclass contains all fields and methods the superclass has. It is unnecessary to duplicate the information for those fields and methods in the subclass metadata.

Second, for external classes defined in other compilation units, the compiler records only information of the fields and methods referenced in this compilation unit. The compilation unit that defines the external class writes full metadata for the class. Checking the consistency of metadata in different compilation units is considered future work because currently the linker discards type information in object files.

Third, Bartok excludes in the metadata useless private fields–those not referenced in the compilation unit where the enclosing class is defined.

3.4 Data

Similar to metadata, object-oriented programs use more data than functional or imperative ones. For each class, the compiler creates a vtable that contains all virtual methods of the class, including the ones from its parent class. Also, the compiler creates a runtime tag for the class, which uniquely identifies the

class at run time. Runtime tags are used for type casts. Vtables and runtime tags are put into the data section of object files.

The checker needs to check the data to see if they match their specified types. For example, we cannot claim an integer in the data section to have a pointer type.

Vtables and runtime tags are complicated data structures. A vtable in Bartok contains a fixed section (84 bytes) and a list of virtual method pointers. The fixed section is the same for all vtables, including a pointer to the runtime tag of the corresponding class, a pointer to the interface method table, an array of pointers to runtime tags of superclasses, a pointer to the runtime tag of the element type if the corresponding class is an array, and many other fields. A runtime tag in Bartok needs 80 bytes, including a pointer to the corresponding vtable, a pointer to the runtime tag of the parent class, interface table information, etc.

The structure of those data is flattened in the object files in the sense that a record of two integers looks exactly the same as two independent integers. When writing a vtable into the data section of object files, Bartok writes first a label indicating the beginning of the vtable, then each piece of information in the fixed section, and at last a list of virtual method pointers. Each part of the vtable is represented as a piece of data, independent of each other. The only requirement is that they have to be adjacent and in order. Runtime tags have similar encodings.

One approach to represent the types for vtables and runtime tags is to record the type for each piece of data in the data structures. Using this elaborate representation for data allows the compiler to choose vtable layout and runtime tag layout. The compiler even has the flexibility to use different layouts for vtables and runtime tags of different classes. But the type information for these large data structures takes significant space.

Instead, Bartok bakes in the layout strategies of vtables and runtime tags. It records only the type of the starting label as the vtable or the runtime tag of the corresponding class. The type information for the following components can be inferred from the layout strategies of vtables and runtime tags and the metadata for the corresponding class. Often an object-oriented compiler uses a uniform layout strategy for all vtables and runtime tags, so this is not a severe restriction for the compiler.

3.5 Integer Encoding

Bartok uses many integers in its type representation. The table-based type sharing encodes component types as integer indices into the type array. Field and method offsets in metadata are integers. Type information for basic blocks and instructions needs to track the locations (offsets from the beginning of the code section) of the basic blocks and the instructions.

Making integer representations concise has a surprisingly big impact on the whole size of type information. Bartok first used a naive four-byte representation for integers, and then changed to a more space-efficient one, reducing the total type size from 64% of code and data size to 36%.

The compressed integer representation is adopted from the one used by CIL for signatures, with slight changes to represent large integers. Integers between 0 and 0x7F are encoded as one-byte integers as they are (the highest bit—bit 7—is 0). Those between 0x80 and 0x3FFF are encoded as two-byte integers with the highest two bits (bits 15 and 14) 10. Integers between 0x4000 and 0x1FFFFFFF are encoded as four-byte integers with the highest three bits (bits 31-29) 110. Other integers are represented as five-byte integers with the first byte 0xFF and the rest four bytes for the integer value. Many integers are represented with fewer than 4 bytes because many type indices and offsets fall into the range between 0 and 0x3FFF.

4 Measurement

This section explains the experimental results and effect of the type representation techniques. Bartok has about 150 benchmarks for day-to-day testing. The table below shows seven large ones and the size of their code and data. The other parts in object files, including relocation information and symbol information, are not counted. Three benchmarks —asmlc, selfhost1421, and lcscbench—are large, code and data size ranging from 2.5MB to 5.4MB. The other four are small, code and data size ranging from 7.7KB to 271KB.

Name	**Description**	**Size of pure code and data** (bytes)
ahcbench	An implementation of Adaptive Huffman Compression.	13,908
asmlc	A compiler for Abstract State Machine Language.	5,436,519
selfhost1421	A version of the Bartok Compiler.	2,507,286
lcscbench	The front end of a C# compiler.	2,944,867
mandelform	An implementation of mandelbrot set computation.	7,702
sat_solver	An implementation of SAT solver written in C#.	135,571
zinger	A model checker to explore the state of the zing model.	271,368

The measurement uses the separate compilation mode of Bartok—user programs are compiled separately from the libraries. The type checker checks the object code compiled from the user programs. We expect that less type information is needed to type check executables than checking object code because metadata for cross reference between compilation units is no longer necessary. Currently if a unit A refers to a class defined in another unit B, the object code for unit A must contain enough metadata for the class. Executables do not need such metadata. The linker just needs to check if the two units have consistent metadata.

4.1 Effect of Type Representation Techniques

With sharing common subterms but no other techniques, the type information can be several times larger than the code and data. Applying our type representation techniques reduces type size from about 2.3 times of code and data size to 36% (geometric means).

Figure 1 summarizes the effect of type representation techniques for the benchmarks. The X axis is the benchmarks. The Y axis is the type size normalized by the code and data size (the code and data size is 1. The lower bars, the better). For each benchmark, the left most bar is the code and data size. The next bar is the type size with only sharing common subterms. Each of the other bars shows the type size with one more technique applied (improve upon its adjacent left bar), till the rightmost bar which shows the type size with all techniques applied. The explanation of individual techniques is as follows.

Code Pointer Type. As explained earlier, code pointer types have the biggest impact on type size. It pays to optimize the representation of code pointer types as much as possible. Simply using function types to encode code pointer types reduces type size from 230% of code and data size to 101%.

Eliding Block Preconditions. Besides using function types, Bartok reduces the number of code pointer types by eliding preconditions of some basic blocks. This technique has a much smaller impact (type size reduced from 102% to 96%) than the previous one although it significantly reduces the number of blocks that need preconditions. The preconditions of about 60% of the basic blocks are removed. The reason for the small impact is that basic blocks in a function tend to share many subterms, such as the type of the return address and the stack (if the stack is the same at the beginning of those blocks). Most merge points still need code pointer types for their preconditions. The subterms are encoded even if many blocks' preconditions that share the subterms are removed.

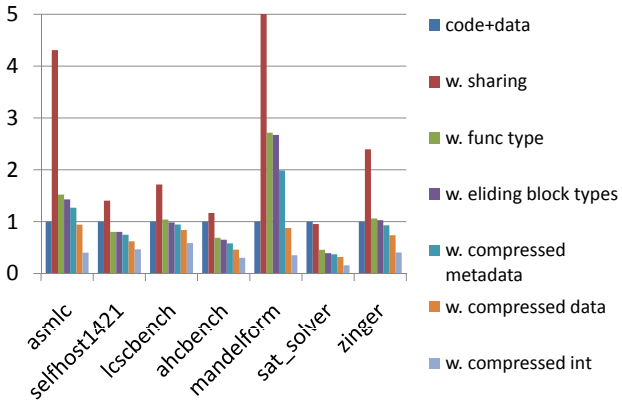


Figure 1: Effect of Type Representation Techniques

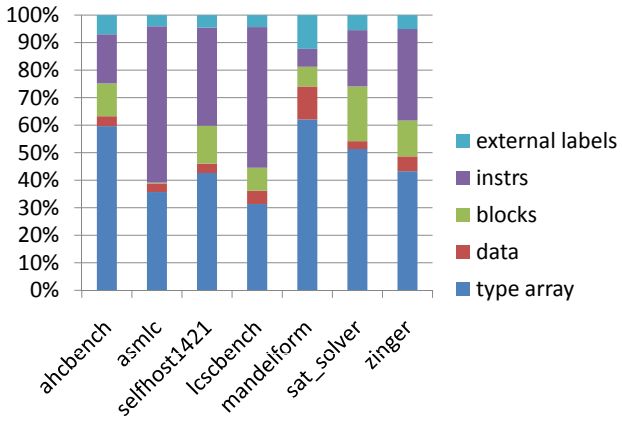


Figure 2: Type Information Breakdown

Compressed Metadata. With efficient representation of code pointer types, we further compress metadata by sharing metadata between superclasses and subclasses and removing unnecessary field or method information. This technique reduces type size from 96% of code and data size to 86%.

Compressed Data. Type size is further reduced to 64% of code and data size after we remove types for small data pieces in vtables and runtime tags. This technique reduces significantly the number of data that need types. Only 14% data need to encode their types after this technique. A large portion of the data section is for vtables and runtime tags.

Compressed Integers. Integer compression reduces the type size from 64% of code and data size to 36%. The reduction is because of the extensive usage of integers in type encoding, such as type indices and locations.

4.2 Type Information Breakdown

Now with all of the discussed techniques applied, let us look into what is in the type section of object files. Figure 2 shows a summary (all in percentage of the whole type size). Note that this shows just an estimate of relative sizes of various type information. The type array contains types for all data, blocks, and instructions. It is not surprising that the type array takes the largest chunk (45% of the type section). Instruction type is the second largest (20%). Type coercions take a significant portion of the instruction types. Types for data take only 4%, the smallest, due to the data compression technique.

4.3 Checking Time

We measure the type checking time to show the efficiency of the type checker. The performance numbers were measured on a PC running Windows XP SP3 with two 3.6GHz CPUs and 2GB of memory. The table below lists the absolute checking time in seconds and the checking time relative to the compilation time. Checking is fast and takes only about 2% of the compilation time. The checking time is roughly proportional to the code and data size, with the exception of the smallest benchmarks (ahcbench and mandelform), which is likely because of measurement accuracy.

Benchmark	**Checking Time**(sec)	**% of Comp Time**
ahcbench	0.11	1.2 %
asmlc	11.4	3.2 %
selfhost1421	4.90	3.6 %
lcscbench	3.13	2.6 %
mandelform	0.08	0.3 %
sat_solver	0.22	1.7 %
zinger	0.59	1.8 %
geomean		1.7 %

5 Related Work

The most relevant work to this paper is the TALx86 compiler, where type information was reduced to about 50% of the object code [7, 6]. The type representation of Bartok differs from that of TALx86 in the following aspects. First, Bartok's source language is object-oriented, thus Bartok has to encode type information for a large amount of data and metadata required by object-oriented programming, such as vtables. Techniques more relevant to object-oriented languages, such as compression of metadata and data, prove to be important for Bartok. Second, Bartok differs from TALx86 in several design choices. It separates type information from code and requires that each basic block be checked only once. The TALx86 type checker may check a basic block multiple times (for different paths). For encoding of code pointer types, Bartok uses function types whereas TALx86 used parameterized abbreviation (type-level functions to abstract common structures of the code pointer types). Parameterized abbreviation may achieve more concise representation of code pointer types. It is considered future work to adopt this technique in Bartok.

Necula *et al.* developed a certifying compiler called SpecialJ for Java [4] as part of the Proof-Carrying Code framework [9]. The compiler translates Java bytecode to x86 assembly code with type annotations, and a safety proof that certifies the code. Later work greatly reduced the size of safety proof [11]: instead of encoding the complete proof, the compiler encoded only hints (oracles) to the proof checker so that the checker can determine which rule to apply when it has multiple choices. Type information and metadata are not included in the measurement of safety proof size.

Necula and Lee also developed a logical framework LF_i that allows reconstructing proof subterms during proof checking, so that those subterms do not need to be encoded in the proof [10]. This results in both compact representation of proofs and efficient proof checking.

Some compression techniques for Java class files reorganized the files, including data, metadata, and code, to achieve better sharing and compression [13, 1]. Many of the techniques, e.g., reference compression, may be applied to our type representation to further reduce type size.

Shao showed a few effective techniques in the FLINT compiler that reduce the memory consumption of types [14]. One of the techniques is hash-consing. The source language of the FLINT compiler is SML, a functional language. There is not much metadata information the compiler needs to track. Also,

the FLINT intermediate language does not need to handle the control stack or code pointer types because the compiler has not exposed the stack yet at this stage.

6 Conclusion

This paper explains how our large-scale object-oriented certifying compiler achieves both concise type representation and efficient type checking for TAL. In our experience, efficient encoding of information specific to object-oriented languages—metadata and data—has a big impact on type information size.

References

[1] Q. Bradley, R. Horspool, and J. Vitek. Jazz: An efficient compressed format for java archive files. In *Proceedings of CASCON '98*, 1998.

[2] J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 183–192, New York, NY, USA, 2008. ACM.

[3] J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *ACM Symposium on Principles of Programming Languages*, pages 38–49, 2005.

[4] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[5] Microsoft Corp. Microsoft portable executable and common object file format specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFFdwn.mspx.

[6] Dan Grossman and Greg Morrisett. Scalable certification for typed assembly language. In *In Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, pages 117–145. Springer-Verlag, 2000.

[7] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[8] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, May 1999.

[9] G. Necula. Proof-Carrying Code. In *ACM Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[10] G. Necula and P. Lee. Efficient representation and validation of proofs. In *Proc. 13th Symp. Logic in Computer Science*, pages 93–104, 1998.

[11] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 142–154, New York, NY, USA, 2001. ACM.

[12] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

[13] William Pugh. Compressing java class files. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 247–258, 1999.

[14] Z. Shao. Implementing typed intermediate language. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 313–323, Baltimore, Maryland, September 1998.

Deriving Safety Cases from Machine-Generated Proofs

Nurlida Basir and Bernd Fischer
ECS, University of Southampton
SO17 1BJ, U.K
(nb206r,b.fischer)@ecs.soton.ac.uk

Ewen Denney
SGT, NASA Ames Research Center
Mountain View, CA 94035, U.S.A
Ewen.W.Denney@nasa.gov

Abstract

Proofs provide detailed justification for the validity of claims and are widely used in formal software development methods. However, they are often complex and difficult to understand, because they use machine-oriented formalisms; they may also be based on assumptions that are not justified. This causes concerns about the trustworthiness of using formal proofs as arguments in safety-critical applications. Here, we present an approach to develop safety cases that correspond to formal proofs found by automated theorem provers and reveal the underlying argumentation structure and top-level assumptions. We concentrate on natural deduction proofs and show how to construct the safety cases by covering the proof tree with corresponding safety case fragments.

1 Introduction

Demonstrating the safety of large and complex software-intensive systems requires marshalling large amounts of diverse information, e.g., models, code or mathematical equations and formulas. Obviously tools supported by automated analyses are needed to tackle this problem. For the highest assurance levels, these tools need to produce a *traceable safety argument* that shows in particular where the code as well as the argument itself depend on any external assumptions but many techniques commonly applied to ensure software safety do not produce enough usable evidence (i.e., justification for the validity of their claims) and can thus not provide any further insights or arguments. In contrast, in formal software safety certification [3], formal proofs are available as evidence. However, these proofs are typically constructed by automated theorem provers (ATPs) based on machine-oriented calculi such as resolution [10]. They are thus often too complex and too difficult to understand, because they spell out too many low-level details. Moreover, the proofs may be based on assumptions that are not valid, or may contain steps that are not justified. Consequently, concerns remain about using these proofs as *arguments* rather than just *evidence* in safety-critical applications. In this paper we address these concerns by systematically constructing safety cases that correspond to formal proofs found by ATPs and explicitly highlight the use of assumptions.

The approach presented here reveals and presents the proof's underlying argumentation structure and top-level assumptions. We work with natural deduction (ND) proofs, which are closer to human reasoning than resolution proofs. We explain how to construct the safety cases by covering the ND proof tree with corresponding safety case fragments. The argument is built in the same top-down way as the proof: it starts with the original theorem to be proved as the top goal and follows the deductive reasoning into subgoals, using the applied inference rules as strategies to derive the goals. However, we abstract away the obvious steps to reduce the size of the constructed safety cases. The safety cases thus provide a "structured reading guide" for the proofs that allows users to understand the claims without having to understand all the technical details of the formal proof machinery. This paper is a continuation of our previous work to construct safety cases from information collected during the formal verification of the code [2], but here we concentrate on the proofs rather than the verification process.

2 Formal Software Safety Certification

Formal software safety certification uses formal techniques based on program logics to show that the program does not violate certain conditions during its execution [3]. A *safety property* is an exact char-

E. Denney, T. Jensen (eds.); The 3rd International Workshop on Proof Carrying Code and Software Certification, pp. 13-17

acterization of these conditions, based on the operational semantics of the programming language. Each safety property thus describes a class of hazards. The safety property is enforced by a *safety policy*, i.e., a set of verification rules that take initial set of safety requirements that formally represent the specific hazards identified by a safety engineer [8], and derive a number of proof obligations. Showing the safety of a program is thus reduced to formally showing the validity of these proof obligations: a program is considered safe wrt. a given safety property if proofs for the corresponding safety proof obligations can be found. Formally, this amounts to showing $D \cup A \models P \Rightarrow C$ for each obligation i.e., the formalization of the underlying *domain theory* D and a set of *formal certification assumptions* A entail a conjecture, which consists of a set of premises P that have to imply the *safety condition* C.

The different parts of these proof obligations have different levels of trustworthiness, and a safety case should reflect this. The hypotheses and the safety condition are inferred from the program by a trusted software component implementing the safety policy, and their construction can already be explained in a safety case [2]. In contrast, both the domain theory and the assumptions are manually constructed artifacts that require particular care. In particular, the safety case needs to highlight the use of assumptions. These have been formulated in isolation by the safety engineer and may not necessarily be justified, and are possibly inconsistent with the domain theory. Moreover, fragments of the domain theory and the assumptions may be used in different contexts, so the safety case must reflect which of them are available at each context. By elucidating the reasoning behind the certification process and drawing attention to potential certification problems, there is less of a need to trust the certification tools, and in particular, the manually constructed artifacts.

3 Converting Natural Deduction Proofs into Safety Cases

Natural deduction [6] systems consist of a collection of proof rules that manipulate logical formulas and transform premises into conclusions. A conjecture is proven from a set of assumptions if a repeated application of the rules can establish it as conclusion. Here, we focus on some of the basic rules; a full exposition of the ND calculus can be found in the literature [6].

Conversion Process. ND proofs are simply trees that start with the conjecture to be proven as root, and have given axioms or assumed hypotheses at each leaf. Each non-leaf node is recursively justified by the proofs that start with its children as new conjectures. The edges between a node and all of its children correspond to the inference rule applied in this proof step. The proof tree structure is thus a representation of the underlying argumentation structure. We can use this interpretation to present the proofs as *safety cases* [7], which are structured arguments as well and represent the linkage between evidence (i.e., the deductive reasoning of the proofs from the assumptions to the derived conclusions) and claims (i.e., the original theorem to be proved). The general idea of the conversion from ND proofs to safety cases is thus fairly straightforward. We consider the conclusion as a goal to be met; the premise(s) become(s) the new subgoal(s). For each inference rule, we define a safety case template that represents the same argumentation. The underlying similarity of proofs and safety cases has already been indicated in [7] but as far as we know, this idea has never been fully explored or even been applied to machine-generated proofs (see Figure 1 for some example rules and templates). Here, we use the Goal Structuring Notation [7] as technique to explicitly represent the logical flow of the proof's argumentation structure.

Implications. The implication elimination follows the general pattern sketched above but in the introduction rule we again temporarily assume A as hypothesis together with the list of other available hypotheses, rather than deriving a proof for it. We then proceed to derive B, and *discharge* the hypothesis by the introduction of the implication. The hypothesis A can be used at given in the prove of B, but the conclusion $A \Rightarrow B$ no longer depends on the hypothesis A after B has been proved. In the safety case fragment, we use a justification to record the use of the hypothesis A, and thus to make sure that the

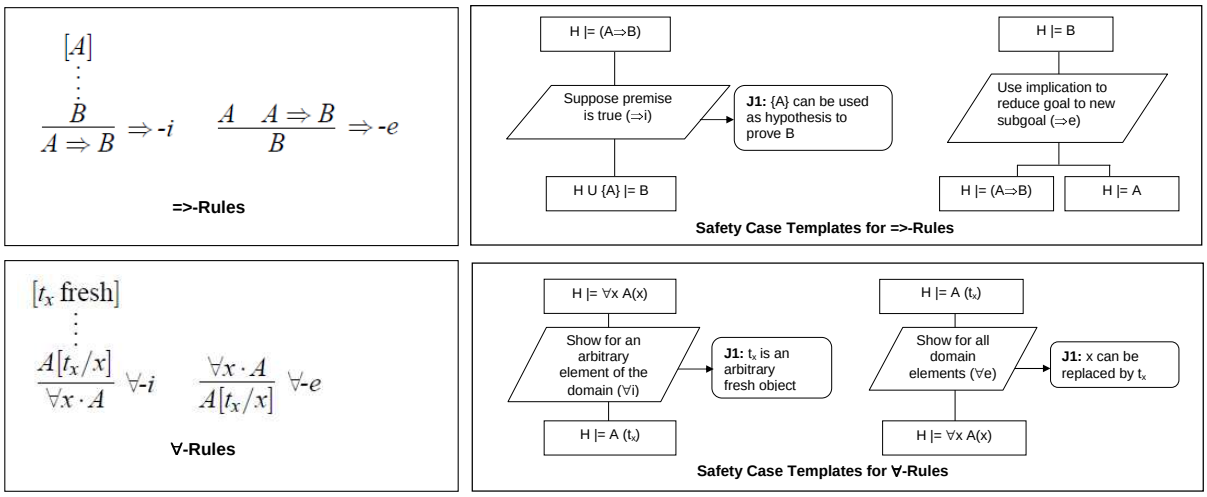


Figure 1: Safety Case Templates for Natural Deduction Rules

introduced hypotheses are tracked properly.

Universal quantifiers. The ND proof rules for quantifiers focus on the replacement of the bound variables with objects and vice versa. For example, in the elimination rule for universal quantifiers, we can conclude the validity of the formula for any chosen domain element t_x. In the introduction rule, however, we need to show it for an arbitary but fresh object t_x (that is, a domain element which does not appear elsewhere in H, A, or the domain theory and assumptions). If we can derive a proof of A, where x is replaced by the object t_x, we can then discharge this assumption by introduction of the quantifier. The safety case fragments record this replacement as justification. The hypotheses available for the subgoals in the $\forall$-rules are the same as those in the original goals.

4 Hypothesis Handling

An automated prover typically treats the domain theory D and the certification assumptions A as premises and tries to derive $\bigwedge(D \cup A) \wedge P \Rightarrow C$ from an empty set of hypotheses. As the proof tree grows, these premises will be turned into hypotheses, using the $\Rightarrow$- introduction rule (see Figure 1). In all other rules, the hypotheses are simply inherited from the goal to the subgoals. However, not all hypotheses will actually be used in the proof, and the safety case should highlight those that are actually used. This is particularly important for the certification assumptions. We can achieve this by modifying the template for the $\Rightarrow$- introduction (see Figure 2a). We can distinguish between the hypotheses that are actually used in the proof of the conclusion (denoted by $A_1, ..., A_k$) and those that are vacuously discharged by the $\Rightarrow$- introduction (denoted by $A_k + 1, .., A_n$). We can thus use two different justifications to mark this distinction. Note that this is only a simplification of the presentation and does not change the structure of the underlying proof, nor the validity of the original goal. It is thus different from using a *relevant implication* [1] under which $A \Rightarrow B$ is only valid if the hypothesis A is actually used.

In order to minimize the number of hypotheses tracked by the safety case, we need to analyze the proof tree from the leaves up, and propagate the hypotheses towards the root. By revealing only these used hypotheses as assumptions, the validity of their use in deriving the proof can be checked more easily. In our work, we also highlight the use of the external certification assumptions that have been formulated in isolation by the safety engineer. For example, in Figure 2b, the hypothesis has_unit(float_7_0e_minus_1, ang_vel), meaning that a particular floating point variable represents an angular velocity, has been speci-

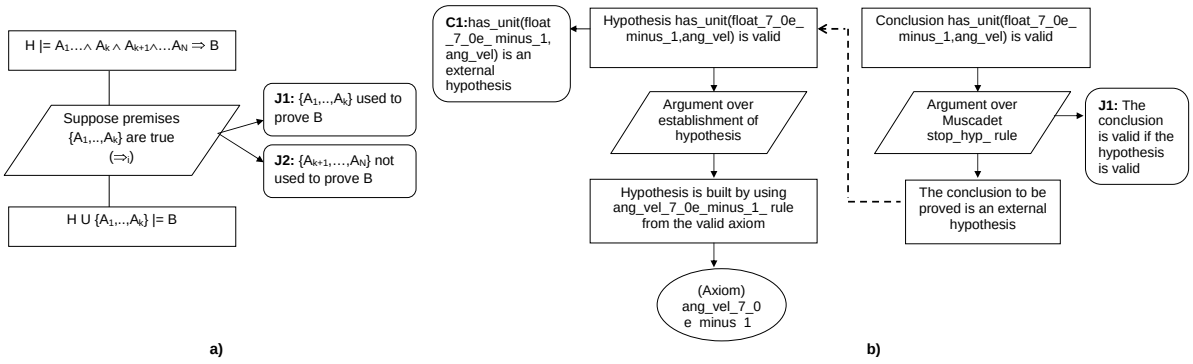


Figure 2: Hypothesis Handling

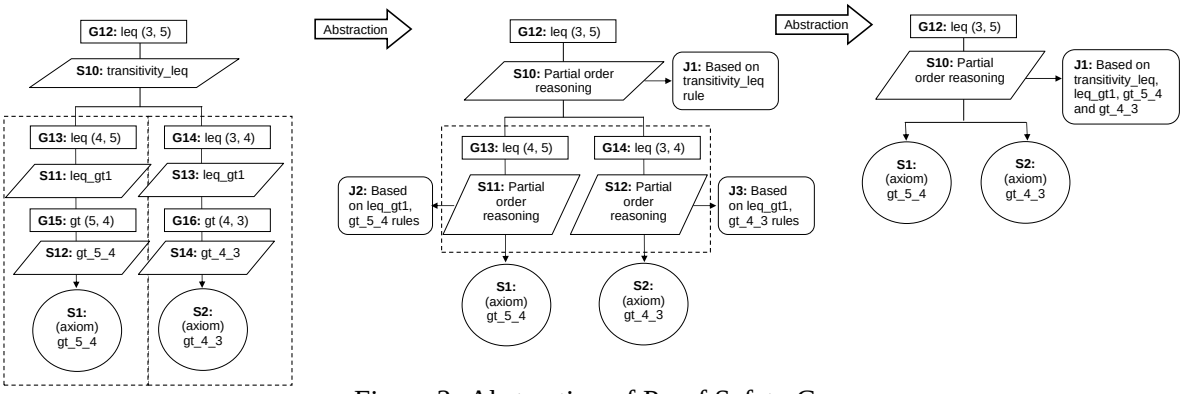


Figure 3: Abstraction of Proof Safety Case

fied as external assumption. This is tracked properly in the safety case, and its role in deriving the proofs can be checked easily.

5 Proof Abstraction

We have applied our approach to proofs found by the Muscadet [9] theorem prover during the formal certification of the frame safety of a component of an attitude control system as an example. Muscadet is based on ND, but to improve performance, it implements a variety of derived rules in addition to the basic rules of the calculus. This includes rules for dedicated equality handling, as well as rules that the system builds from the definitions and lemmas, and that correspond the application of the given definitions and lemmas. While these rules make the proofs shorter, their large number makes the proofs also in turn more difficult to understand. This partially negates the original goal of using a ND prover. We thus plan to optimize the resulting proofs by removing some of the book-keeping rules (e.g., return_proof) that are not central to the overall argumentation structure. Similarly, we plan to collapse sequences of identical book-keeping rules into a single node. In general, however, we try to restructure the resulting proof presentation to help in emphasizing the essential proof steps. In particular, we plan to group sub-proofs that apply only axioms and lemmas from certain obvious parts of the domain theory (e.g., ground arithmetic or partial order reasoning) and represent them as a single strategy application. Figure 3 shows an example of this. Here, the first abstraction step collapses the sequences rooted in G13 and G14, noting the lemmas which had been used as strategies as justifications, but keeping the branching that is typical for the transitivity. A second step then abstracts this away as well.

6 Conclusions

We have described an approach whereby a safety case is used as a "structured reading" guide for the safety proofs. Here, assurance is not implied by the trust in the ATPs but follows from the constructed argument of the underlying proofs. However, the straightforward conversion of ND proofs into safety cases turn out to be far from satisfactory as the proofs typically contain too many details. In practice, a superabundance of such details is overwhelming and unlikely to be of interest anyway so careful use of abstraction is needed [5].

The work we have described here is still in progress. So far, we have automatically derived safety cases for the proofs found by Muscadet prover [9]. This work complements our previous work [2] where we used the high-level structure of annotation inference to explicate the top-level structure of such software safety cases. We consider the safety case as a first step towards a fully-fledged software certificate management system [4]. We also believe that our research will result in a comprehensive safety case (i.e., for the program being certified the safety logic, and the certification system) that will clearly communicate the safety claims, key safety requirements, and evidence required to trust the software safety.

Acknowledgements. This material is based upon work supported by NASA under awards NCC2-1426 and NNA07BB97C. The first author is funded by the Malaysian Government, IPTA Academic Training Scheme.

References

[1] A.R. Anderson and N. Belnap. *Entailment: the logic of relevance and necessity*. Princeton University Press, 1975.

[2] N. Basir, E. Denney, and B. Fischer. Constructing a Safety Case for Automatically Generated Code from Formal Program Verification Information. *In SAFECOMP'08*, pages 249–262, 2008.

[3] E. Denney and B. Fischer. Correctness of Source-Level Safety Policies . In *Proc. FM 2003: Formal Methods*, 2003.

[4] E. Denney and B. Fischer. Software Certification and Software Certificate Management Systems (position paper). *Proceedings of the ASE Workshop on Software Certificate Management Systems (SoftCeMent '05)*, pages 1–5, 2005.

[5] E. Denney, J. Power, and K. Tourlas. Hiproofs: A Hierarchical Notion of Proof Tree. In *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, volume 155, pages 341 – 359, 2006.

[6] M. Huth and M. Ryan. *Logic in Computer Science Modelling and Reasoning about Systems*, volume 2nd Edition. Cambridge University Press, 2004.

[7] T. P. Kelly. *Arguing Safety - A Systematic Approach to Managing Safety Cases*. PhD thesis, University of York, 1998.

[8] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[9] D. Pastre. MUSCADET 2.3: A Knowledge-Based Theorem Prover Based on Natural Deduction. In *IJCAR*, pages 685–689, 2001.

[10] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *ACM*, 1965.

PCC Framework for Program-Generators *

Soonho Kong Wontae Choi Kwangkeun Yi
Seoul National University
{soon,wtchoi,kwang}@ropas.snu.ac.kr

Abstract

In this paper, we propose a proof-carrying code framework for program-generators. The enabling technique is abstract parsing, a static string analysis technique, which is used as a component for generating and validating certificates. Our framework provides an efficient solution for certifying program-generators whose safety properties are expressed in terms of the grammar representing the generated program. The fixed-point solution of the analysis is generated and attached with the program-generator on the code producer side. The consumer receives the code with a fixed-point solution and validates that the received fixed point is indeed a fixed point of the received code. This validation can be done in a single pass.

1 Introduction

To certify the safety of a mobile program-generator, we need to ensure not only the safe execution of the generator itself but also that of the generated programs. Safety properties of the generated programs are specified efficiently in terms of the grammar representing the generated programs. For instance, the safety property "generated programs should not have nested loops" can be specified and verified by the reference grammar for the generated programs.

Recently, Doh, Kim, and Schmidt presented a powerful static string analysis technique called abstract parsing [4]. Using LR parsing as a component, abstract parsing analyzes the program and determines whether the strings generated in the program conform to the given grammar or not.

In this paper, we propose a Proof-Carrying Code (PCC) framework [8, 9] for program-generators. We adapt abstract parsing to check the generated programs of the program-generators. With the grammar specifying the safety property of the generated programs, the code producer abstract-parses the program-generator and computes a fixed-point solution as a certificate. The code producer sends the program-generator with the computed fixed-point solution. The code consumer receives the program-generator accompanied with the fixed-point solution and validates that the received fixed point is indeed the solution for the received program-generator. Our framework can be seen as an abstraction-carrying code framework [1, 5] specialized to program-generators which is modeled by a two-staged language with concatenation.

This work is, to our knowledge, the first to present a proof-carrying code framework that certifies grammatical properties of the generated programs. Directly computing the parse stack information as a form of the fixed-point solution, abstract parsing provides an efficient way to validate the certificates on the code consumer side. In contrast to abstract parsing, the previous static string analysis techniques [3, 7, 2] approximate the possible values of a string expression of the program with a grammar and see whether the approximated grammar is included in the reference grammar. This grammar inclusion check takes too much time and makes those techniques difficult to be used as a validation component of a PCC framework.

E. Denney, T. Jensen (eds.): The 3rd International Workshop on Proof Carrying Code and Software Certification, volume 0, issue: 0, pp. 18-22

*This work was supported by the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2009 and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF). (R11-2008-007-01002-0).

2 Language

For the further development of our idea, we consider a two-staged language with concatenation in which program-generators can be modeled. The language is an imaginary, first-order language whose only value is code. The language is minimal, so as not to distract our focus on static analysis. For example, loops and conditional jumps are without the condition expression, for which abstract interpretation anyway considers all iterations and all branches.

A program is an expression e:

$$e \in Exp ::= x \mid \texttt{let}\, x\, e_1\, e_2 \mid \texttt{or}\, e_1\, e_2 \mid \texttt{re}\, x\, e_1\, e_2\, e_3 \mid \text{`}f$$

An expression can contain code fragments f:

$$f \in Frag ::= x \mid \texttt{let} \mid \texttt{or} \mid \texttt{re} \mid \texttt{(} \mid \texttt{)} \mid f_1.f_2 \mid \texttt{,}e$$

Operational semantics of the language is defined in Figure 3 (left).

Expression $\texttt{or}\, e_1\, e_2$ is for branches. It could be the value of e_1 or the value of e_2. Expression $\texttt{re}\, x\, e_1\, e_2\, e_3$ is for loops. Variable x has the value of e_1 as its initial value. Loop body e_2 is iterated ≥ 0 times. The result of each iteration e_2 will be bound to x in e_2 for next iteration or in e_3 for the result of the loop. Backquote form $\text{`}f$ is for code fragment $f$. We construct the fragment by using the following tokens: variables, `let`, `or`, `re`, `(`, and `)`. Compound fragment $f_1.f_2$ concatenates two code fragments $f_1$ and $f_2$. Comma fragment $\texttt{,}e$ first evaluates $e$ then substitutes its result code value for itself. Note that the meaning of $\text{`}f$ and $\texttt{,}e$ is the same as in LISP's quasi-quotation system.

3 Abstract Parsing

In our framework, we use abstract parsing [4] as a component to generate and validate the certificate. Abstract parsing derives data-flow equations from the program and solves them in the parsing domain. In [6], we formulated abstract parsing in the abstract interpretation framework.

The key idea of abstract parsing is an abstraction of code. Code c is abstracted into a parse-stack transition function $f = \lambda p.parse(p,c)$ where *parse* is a parsing function defined by an LR parser generator with the safety grammar G. This choice of abstraction is necessary to handle code concatenation $x.y$. If abstracted functions for the code fragments x and y are $f_x = \lambda p.parse(p,x)$ and $f_y = \lambda p.parse(p,y)$ respectively, an abstracted function for the code concatenation $x.y$ is constructed by function composition of f_x and f_y as $f_{x.y} = f_y \circ f_x$.

As illustrated in Figure 1, we take a series of abstraction steps for the value domain of the semantics.

Collecting Semantics	⟶	Concrete Parsing Semantics	⟶	First Step Abstraction Semantics	⟶	Parameterized Abstract Parsing Semantics
2^{Code}		$2^{P \to P}$		$2^P \to 2^P$		$D^\sharp \to D^\sharp$

Figure 1: Series of abstraction steps for the value domain in semantics where P is the set of parse stacks.

Starting from the collecting semantics defined in Figure 3 (middle), each abstraction of the value domain derives new abstract semantics.

To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain 2^P. Instead of using a particular abstract domain for 2^P, we parameterize this abstract domain by providing conditions which an abstract domain $D^\sharp$ needs to satisfy.

1. $D^\sharp$ should be a complete partial order (CPO).
2. $D^\sharp$ is Galois connected with the set of parse stacks 2^P.
3. An abstracted parsing function $Parse\_action^\sharp$ is defined as a sound approximation of the parsing function $Parse\_action$ which is defined by the LR parser generator with the safety grammar G.

Finally, we derive the abstract parsing semantics for $D^\sharp$ as in Figure 3 (right).

Given a program-generator e and an empty environment σ_0, the analysis computes $F = [\![e]\!]^0_{D^\sharp}\sigma_0$ which is of type $D^\sharp \to D^\sharp$. To determine whether the programs generated by a program-generator e conform to the safety grammar, we check that the following equation holds:

$$F(\alpha_{2^P \to D^\sharp}(\{p_{init}\})) = \alpha_{2^P \to D^\sharp}(\{p_{acc}\})$$

where p_{init} and p_{acc} are the initial parse stack and accepting parse stack for the safety grammar G.

4 PCC Framework for Program-Generators

Figure 2 illustrates a PCC framework for program-generators, an abstraction-carrying code framework [1, 5] specialized to program-generators by means of abstract parsing. The code producer and code consumers share the safety grammar which specifies the safety properties of the generated programs.

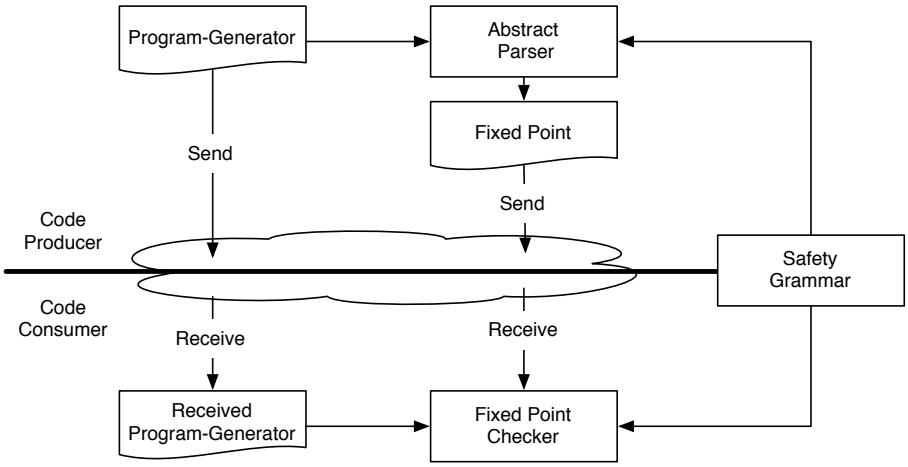


Figure 2: A proof-carrying code framework for program-generators.

The code producer proves the safety of the program-generator by abstract parsing with the shared safety grammar. In a complex and iterative process, the analysis computes a fixed-point solution. This solution is used as a certificate for the safety of the program-generator. The code producer uploads or sends the program-generator with the computed fixed-point solution.

The code consumer downloads or receives the untrusted program-generator and its attached fixed-point solution. The code consumer validates that the received fixed-point solution is indeed a fixed-point solution of the received program-generator. In contrast to the computing a fixed-point solution on the code producer side, checking can be done in a single pass.

5 Issues

The proposed framework addresses two fundamental PCC issues.

1. The certificate, a fixed-point solution for the program-generator, is generated automatically by abstract parsing.
2. Checking procedure on the code consumer side is done efficiently by validating the received fixed-point solution.

However, we have several issues for further investigation.

1. Size of the certificate: We are not sure that the size of the fixed-point solution which our framework generates is small enough for the mobile platform. However, there are some ideas on reducing the size of certificates. First, the certificate can be compressed. Abstract parsing uses an abstract parse stack as a component of the value domain. Since a parse stack is a string of characters from a pre-defined finite alphabet, an appropriate compression algorithm can be used to reduce the size of fixed-point solution. Second, some parts of the certificate could be deleted as long as their recovery takes linear time to the size of the received code.
2. Size of the trust base: Similar to other abstraction-carrying code frameworks, the certificate checker of our framework is almost as complex as the certificate generator. It is essential to simplify the certificate checker to reduce the size of the trust base.

References

[1] E. Albert, G. Puebla, and M. Hermenegildo. Abstract interpretation-based approach to mobile code safety. In *Proceedings of Compiler Optimization meets Compiler Verification*, 2004.

[2] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A practical string analyzer by the widening approach. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, volume 4729 of *Lecture Notes in Computer Science*, pages 374–388, Sydney, Austrailia, November 2006. Springer-Verlag.

[3] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the Static Analysis Symposium*, pages 1–18. Springer-Verlag, 2003.

[4] Kyung-Goo Doh, Hyunha Kim, and David Schmidt. Abstract parsing: static analysis of dynamically generated string output using LR-parsing technology. In *Proceeeding of the International Static Analysis Symposium*, 2009. Available from `http://santos.cis.ksu.edu/schmidt/dohsas09.pdf`.

[5] Manuel V. Hermenegildo, Elvira Albert, Pedro López-García, and Germán Puebla. Abstraction carrying code and resource-awareness. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 1–11, New York, NY, USA, 2005. ACM.

[6] Soonho Kong, Wontae Choi, and Kwangkeun Yi. Abstract parsing for two-staged languages with concatenation. In *Proceeeding of the International Conference on Generative Programming and Component Engineering*, 2009. Available from `http://ropas.snu.ac.kr/~soon/paper/gpce09.pdf`.

[7] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International Conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005. ACM.

[8] George C. Necula. Proof-carrying code. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[9] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, New York, NY, USA, 1998. ACM.

$$\sigma \in Env = Var \to Code$$
$$v \in Code = Token\,sequence$$
$$e \in Exp$$
$$f \in Frag$$

$\boxed{\sigma \vdash^0 e \Rightarrow v}$

$$\frac{}{\sigma \vdash^0 x \Rightarrow \sigma(x)} \quad \text{(variable)}$$

$$\frac{\sigma \vdash^0 e_1 \Rightarrow v \qquad \sigma[x \mapsto v] \vdash^0 e_2 \Rightarrow v'}{\sigma \vdash^0 \texttt{let}\, x\, e_1\, e_2 \Rightarrow v'} \quad \text{(let binding)}$$

$$\frac{\sigma \vdash^0 e_1 \Rightarrow v}{\sigma \vdash^0 \texttt{or}\, e_1\, e_2 \Rightarrow v} \qquad \frac{\sigma \vdash^0 e_2 \Rightarrow v}{\sigma \vdash^0 \texttt{or}\, e_1\, e_2 \Rightarrow v} \quad \text{(branch)}$$

$$\frac{\sigma \vdash^0 e_1 \Rightarrow v \qquad \sigma[x \mapsto v] \vdash^0 loop\, x\, e_2\, e_3 \Rightarrow v'}{\sigma \vdash^0 \texttt{re}\, x\, e_1\, e_2\, e_3 \Rightarrow v'} \quad \text{(loop)}$$

$$\frac{\sigma \vdash^0 e_2 \Rightarrow v \qquad \sigma[x \mapsto v] \vdash^0 loop\, x\, e_2\, e_3 \Rightarrow v'}{\sigma \vdash^0 loop\, x\, e_2\, e_3 \Rightarrow v'} \qquad \frac{\sigma \vdash^0 e_3 \Rightarrow v}{\sigma \vdash^0 loop\, x\, e_2\, e_3 \Rightarrow v}$$

$$\frac{\sigma \vdash^1 f \Rightarrow v}{\sigma \vdash^0 \text{`}\, f \Rightarrow v} \quad \text{(back quote)}$$

$\boxed{\sigma \vdash^1 f \Rightarrow v}$

$$\frac{}{\sigma \vdash^1 x \Rightarrow x} \qquad \frac{}{\sigma \vdash^1 \texttt{let} \Rightarrow \texttt{let}} \qquad \frac{}{\sigma \vdash^1 \texttt{or} \Rightarrow \texttt{or}} \quad \text{(token)}$$

$$\frac{}{\sigma \vdash^1 \texttt{re} \Rightarrow \texttt{re}} \qquad \frac{}{\sigma \vdash^1 \texttt{(} \Rightarrow \texttt{(}} \qquad \frac{}{\sigma \vdash^1 \texttt{)} \Rightarrow \texttt{)}}$$

$$\frac{\sigma \vdash^1 f_1 \Rightarrow v_1 \qquad \sigma \vdash^1 f_2 \Rightarrow v_2}{\sigma \vdash^1 f_1.f_2 \Rightarrow v_1 v_2} \quad \text{(concatenation)}$$

$$\frac{\sigma \vdash^0 e \Rightarrow v}{\sigma \vdash^1 ,e \Rightarrow v} \quad \text{(comma)}$$

Operational semantics of the language.

$$Code = Token\,sequence$$
$$\sigma \in Env = Var \to Code$$
$$[\![e]\!]^0 \in 2^{Env} \to 2^{Code} \qquad \text{(stage-0 semantics)}$$
$$[\![f]\!]^1 \in 2^{Env} \to 2^{Code} \qquad \text{(stage-1 semantics)}$$

$$[\![x]\!]^0\Sigma = \{\sigma(x) \mid \sigma \in \Sigma\}$$
$$[\![\texttt{let}\, x\, e_1\, e_2]\!]^0\Sigma = \bigcup_{\sigma\in\Sigma} \bigcup_{c\in[\![e_1]\!]^0\{\sigma\}} [\![e_2]\!]^0\{\sigma[x \mapsto c]\}$$
$$[\![\texttt{or}\, e_1\, e_2]\!]^0\Sigma = [\![e_1]\!]^0\Sigma \cup [\![e_2]\!]^0\Sigma$$
$$[\![\texttt{re}\, x\, e_1\, e_2\, e_3]\!]^0\Sigma = \bigcup_{\sigma\in\Sigma} [\![e_3]\!]^0\{\sigma[x \mapsto c] \mid c \in fix\lambda C.[\![e_1]\!]^0\{\sigma\} \cup [\![e_2]\!]^0\{\sigma[x \mapsto c'] \mid c' \in C\}\}$$
$$[\![\text{`}\, f]\!]^0\Sigma = [\![f]\!]^1\Sigma$$
$$[\![x]\!]^1\Sigma = \{x\}$$
$$[\![\texttt{let}]\!]^1\Sigma = \{\texttt{let}\}$$
$$[\![\texttt{or}]\!]^1\Sigma = \{\texttt{or}\}$$
$$[\![\texttt{re}]\!]^1\Sigma = \{\texttt{re}\}$$
$$[\![\texttt{(}]\!]^1\Sigma = \{\texttt{(}\}$$
$$[\![\texttt{)}]\!]^1\Sigma = \{\texttt{)}\}$$
$$[\![f_1.f_2]\!]^1\Sigma = \bigcup_{\sigma\in\Sigma} \{xy \mid x \in [\![f_1]\!]^1\{\sigma\} \wedge y \in [\![f_2]\!]^1\{\sigma\}\}$$
$$[\![,e]\!]^1\Sigma = [\![e]\!]^0\Sigma$$

Collecting semantics of the language.

$$\sigma \in Env_{D^\sharp} = Var \to V^\sharp$$
$$[\![e]\!]^0_{D^\sharp} \in Env_{D^\sharp} \to V^\sharp \qquad \text{(stage-0 abstract semantics)}$$
$$[\![f]\!]^1_{D^\sharp} \in Env_{D^\sharp} \to V^\sharp \qquad \text{(stage-1 abstract semantics)}$$

$$[\![x]\!]^0_{D^\sharp}\sigma = \sigma(x)$$
$$[\![\texttt{let}\, x\, e_1\, e_2]\!]^0_{D^\sharp}\sigma = [\![e_2]\!]^0_{D^\sharp}(\sigma[x \mapsto [\![e_1]\!]^0_{D^\sharp}\sigma])$$
$$[\![\texttt{or}\, e_1\, e_2]\!]^0_{D^\sharp}\sigma = [\![e_1]\!]^0_{D^\sharp}\sigma \sqcup [\![e_2]\!]^0_{D^\sharp}\sigma$$
$$[\![\texttt{re}\, x\, e_1\, e_2\, e_3]\!]^0_{D^\sharp}\sigma = [\![e_3]\!]^0_{D^\sharp}(\sigma[x \mapsto fix\,\lambda k.[\![e_1]\!]^0_{D^\sharp}\sigma \sqcup [\![e_2]\!]^0_{D^\sharp}(\sigma[x \mapsto k])])$$
$$[\![\text{`}\, f]\!]^0_{D^\sharp}\sigma = [\![f]\!]^1_{D^\sharp}\sigma$$
$$[\![t]\!]^1_{D^\sharp}\sigma = \lambda D.Parse\_action^\sharp(D,t)$$
$$[\![f_1.f_2]\!]^1_{D^\sharp}\sigma = [\![f_2]\!]^1_{D^\sharp}\sigma \circ [\![f_1]\!]^1_{D^\sharp}\sigma$$
$$[\![,e]\!]^1_{D^\sharp}\sigma = [\![e]\!]^0_{D^\sharp}\sigma$$

Abstract parsing semantics of the language.

Figure 3: Operational semantics, collecting semantics, and abstract parsing semantics of the language.

Assurance Cases for Proofs as Evidence

Sagar Chaki Arie Gurfinkel Kurt Wallnau Charles Weinstock
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, PA
{chaki|arie|kcw|weinstock}@sei.cmu.edu

Abstract

Proof-carrying code (PCC) provides a "gold standard" for establishing formal and objective confidence in program behavior. However, in order to extend the benefits of PCC – and other formal certification techniques – to realistic systems, we must establish the correspondence of a mathematical proof of a program's semantics and its actual behavior. In this paper, we argue that assurance cases are an effective means of establishing such a correspondence. To this end, we present an assurance case pattern for arguing that a proof is free from various proof hazards. We also instantiate this pattern for a proof-based mechanism to provide evidence about a generic medical device software.

1 Introduction

Today's information-based society is dependent on software for its well-being. Software is ubiquitous and invisible in everything from entertainment to critical infrastructure; "out of sight, out of mind" describes current public sentiment about this dependence. Moreover, software components are being interconnected in ways that were never anticipated, or in some cases intended. The adverse effects of software failures resulting from this increased coupling are difficult to contain. Software development has become a commodity service, and software supply chains span the globe; the provenance of any complex software package is, and will likely remain, unknown. Thus, there is an urgent and well recognized need for justifiable confidence that software will behave as intended by the consumer. Moreover, the source of this confidence must be the software artifact itself, and not the identity of, or the processes used by, the software producer. Known provenance and processes are useful, but are not always available to consumers, and do not guarantee acceptable behavior.

Proof-carrying code (PCC) [9] is a "gold standard" for establishing justifiable confidence in program behavior, and has been the epicenter of many recent technical advancements. For example, Chaki et al. have developed [3] a certifying model checker (CMC) and associated machinery to produce PCC against any linear temporal logic (LTL) specification. However, in order to extend the benefits of PCC, and other formal technologies, to large complex systems, we must establish correspondence of a mathematical proof within a *formal system* and the behavior that is exhibited in the *real world*. In this paper, we argue that assurance cases [5] (or cases, for short) provide an effective solution to this correspondence problem. An assurance case is a structured argument that a claimed system-level property has been achieved. Assurance cases employ defeasible reasoning, where a premise (ultimately, evidence) *usually* implies a conclusion. Defeasible reasoning offers an intermediate ground between formal notions of soundness and completeness and the intrinsic uncertainty and incompleteness of any large scale, complex system.

We present an assurance case pattern for arguing that any formal proof is free from various hazards to proof validity. Our pattern handles proof hazards arising from the *use* of the formal technology (did we model the right behavior?), as well as from the technology *itself* (do we trust the theorem prover?). Our approach has several benefits. First, it captures, in pattern form, a variety of threats to the validity of any formal evidence, in effect normalizing and improving the quality of such evidence. Second, the pattern can be extended to argue about the benefits of specific technologies, for example to show why PCC allows us to eliminate model checkers, theorem provers, and even compilers from the trusted computing base. Finally, case patterns and their instances are amenable to being expressed in precise notation, recorded, shared, reviewed, and revised. We demonstrate the effectiveness of our case pattern by instantiating it for

E. Denney, T. Jensen (eds.): The 3rd International Workshop on Proof Carrying Code and Software Certification, volume 0, issue: 0, pp. 23-28

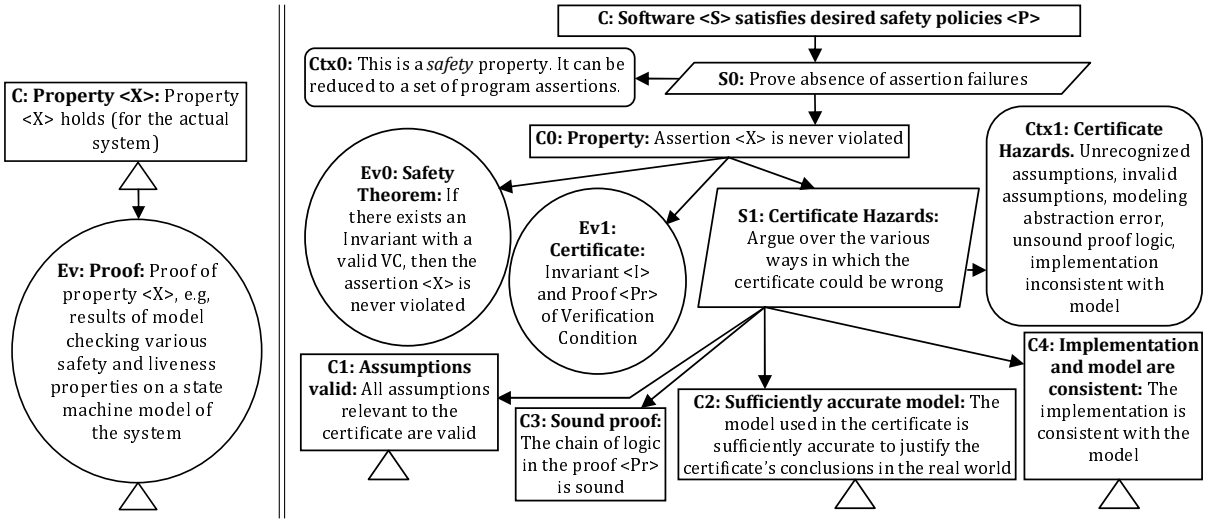


Figure 1: (Left) GSN notation; (Right) top-level GIP assurance case pattern.

a specific application of CMC and PCC technology to provide evidence about software in a hypothetical infusion pump. Our results are preliminary, but encouraging. We believe that, ultimately, such use of cases improves the transitionability of formal techniques to practical situations.

2 Assurance Cases and Infusion Pump Scenario

An assurance case uses a claims-argument-evidence structure to demonstrate the truth of some assertion. It consists of a top-level claim supported by subclaims. Each subclaim is further decomposed into sub-subclaims, and so on, until a claim is directly supported by evidence, i.e., data that is sufficient to support a claim without further argument. Typical examples of evidence are test results, analyses, information about the competency of personnel, etc. The quality of the case (i.e., its soundness and the extent to which it is convincing in supporting its top-level claim) depends on the claim structure and the quality of the presented evidence.

An assurance case is an example of defeasible reasoning, i.e., reasoning where "the corresponding argument is rationally compelling but not deductively valid ... the relationship of support between premises and conclusion is a tentative one, potentially defeated by additional information" [10]. The logical form of a defeasible inference is: **if** E **then**(**usually**) C **unless** $R, S, T,$ **etc**. In other words, claim C follows from evidence E, unless this inference is invalidated by deficiencies R, S, T, etc. The set of deficiencies is never completely known. Even if we argue $\neg R$, $\neg S$, and $\neg T$, new information (e.g., U) could invalidate the $E \Rightarrow C$ inference, or the demonstration of, say, $\neg R$. Therefore, confidence in C is improved by capturing as many deficiencies as possible, and showing their absence.

Infusion Pump Scenario. An infusion pump infuses fluids, medication or nutrients into a patient's circulatory system. Our case study involves a Generalized Infusion Pump (GIP), which includes a built-in drug library. The drug library contains a list of drugs, and, for each drug, the following: (a) drug name, (b) drug concentration, and (c) for each clinical setting, the soft (and hard) minimum (and maximum) allowed infusion rates. The acceptable infusion rate in an emergency environment may be significantly higher than that in a patient room. The acceptable infusion rate for an adult may be significantly higher than for an infant. The GIP consults the drug library when the caregiver is programming an infusion.

We assume the following scenario: (i) the GIP uses an established software and hardware architec-

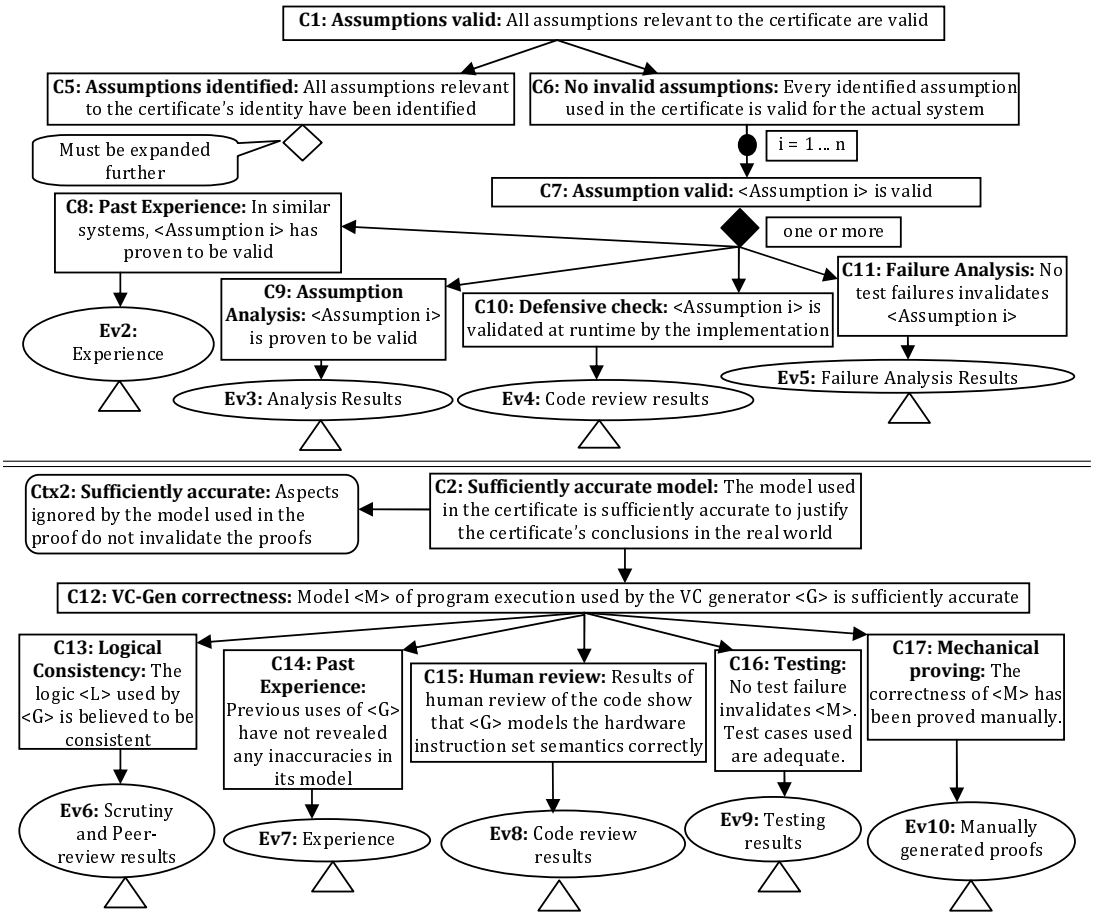


Figure 2: Case patterns for "assumptions valid" (top) and "sufficiently accurate model" (bottom).

ture, (ii) the GIP software is supplied by third parties, and (iii) the GIP manufacturer requires certifiable assurance that the delivered GIP software satisfies the following three (publicly specified) safety policies: **(P1)** if the infusion rate of the selected drug is within the soft bounds appropriate to the setting, the GIP accepts the programming; **(P2)** if the infusion rate is outside of the soft bounds but within the hard bounds the GIP accepts the programming only after a warning and a required override by the caregiver; **(P3)** the GIP cannot be programmed with an infusion rate outside of the hard bounds.

3 GIP Assurance Case Pattern and Instantiation

We use the graphical goal structuring notation (GSN) [5] to express assurance cases. Fig. 1(left) shows, in GSN, the case that "property <X>" holds because there is a proof of the property. Specifically, "property <X> holds" is the claim, and "Proof of property <X>" is the evidence presented in support of this claim. A rectangle indicates a claim, always phrased as a predicate. A circle (or ellipse) indicates evidence (always stated in a noun phrase), and the arrow linking the claim to the evidence implies that the claim is supported by the evidence. The little triangles at the bottom of the rectangle and circle indicate that the claim and evidence are generic and need to be instantiated when this pattern is applied. Angled brackets (<>) characterize what is to be instantiated. In the remaining cases, we omit such triangles when there is an explicit <X> to be instantiated. Also, we use the following additional GSN features. A parallelogram

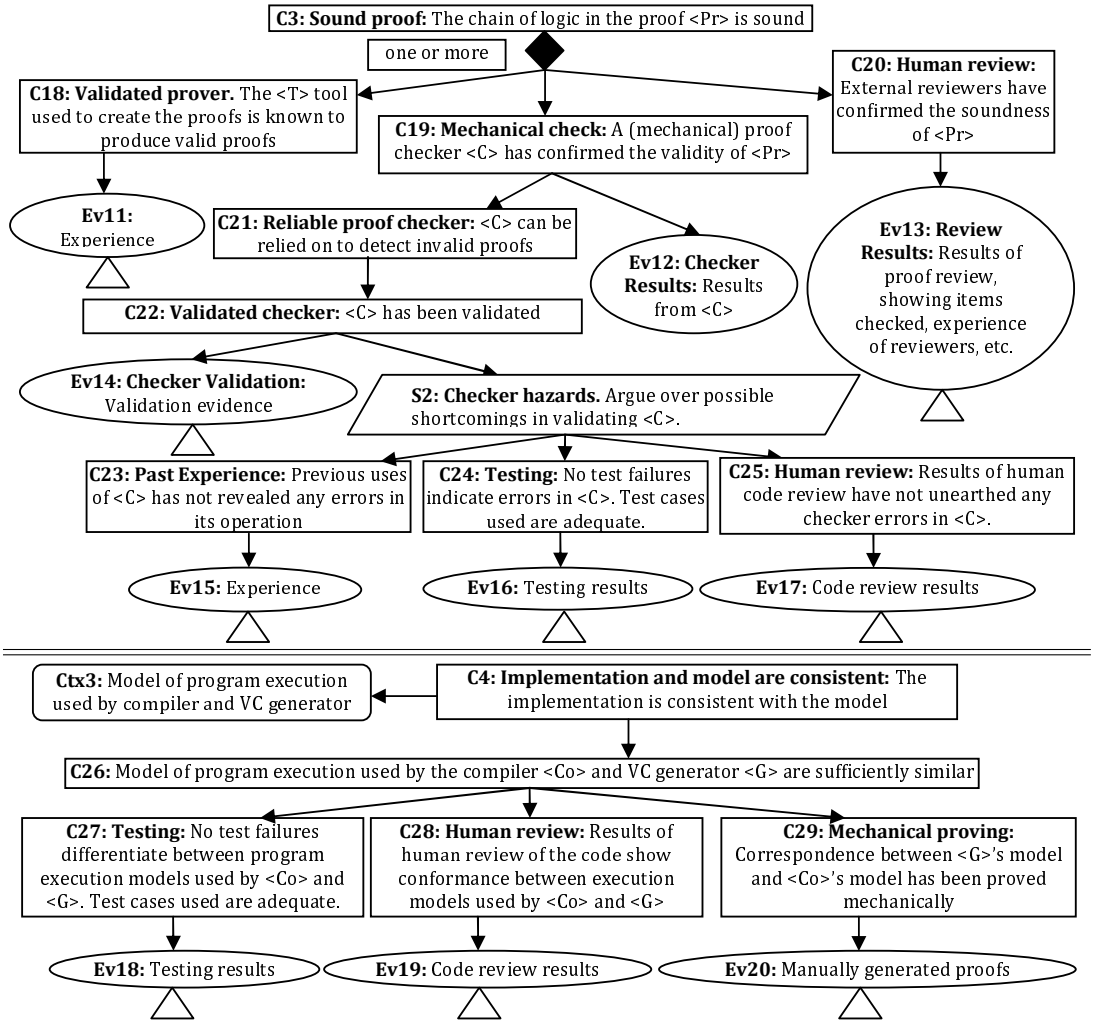


Figure 3: Case patterns for "sound proof" (top) and "implementation and model are consistent" (bottom).

refers to a strategy, while a rounded rectangle refers to a context. Empty diamonds refer to parts that have been left out, but must be expanded further. Solid diamonds refer to a choice between various alternatives. A solid circle denotes iteration.

Fig. 1(right) shows, in GSN, the top-level assurance case pattern for the generic claim "Software <S> satisfies desired safety policies <P>". It leaves the following four sub-claims to be expanded further: **(C1)** assumptions valid, **(C2)** sufficiently accurate model, **(C3)** sound proof, and **(C4)** implementation and model are consistent. The case pattern for **(C1)** and **(C2)** are shown in Fig. 2. Note that the case for **(C1)** has a sub-claim "assumptions identified" that we do not expand further for brevity. The case patterns for **(C3)** and **(C4)** are shown in Fig. 3.

Certification Mechanism. We consider a specific certification mechanism, called PCCCMC, that uses a combination of PCC and CMC to provide formal evidence of safe runtime behavior of programs [3]. The input to PCCCMC is a C program P containing an assertion ASRT. The output is a proof-certificate consisting of an invariant INVAR and a proof PROOF. Let P be the GIP software such that ASRT enforces the desired safety policies **P1–P3**. Then a run of PCCCMC on P consists of the following steps: (i) INVAR is generated using a certifying software model checker CMC; (ii) a verification condition VC is

generated using weakest preconditions by a VCGEN tool; intuitively, VC is a logical formula in a suitable logic $\mathscr{L}$ expressing that INVAR is inductive and implies ASRT; (iii) PROOF is generated by checking the validity of VC using a proof-generating theorem prover PROVER, (iv) PROOF is checked via a CHECKER. The correctness of PCCCMC relies on the "safety theorem" which basically states that *P* does not violate ASRT at runtime if there exists an INVAR for which the VC is valid.

Pattern Instantiation. We now instantiate our assurance case patterns in the context of PCCCMC. In the top-level pattern (see Fig. 1) we instantiate `S` with the GIP Software, `P` with **P1–P3**, and `X` with ASRT. Also, we instantiate `I` with INVAR, and `P` by PROOF. In the pattern for **C1**, we identify and instantiate as many assumptions as possible that are relevant to the certificate. In the pattern for **C2**, we instantiate `G` by VCGEN, `M` by the execution semantics of the GIP Software used by VCGEN, and `L` by $\mathscr{L}$. In the pattern for **C3**, we instantiate `P` by PROOF, `T` by PROVER, and `C` by CHECKER. Finally, in the pattern for **C4**, we instantiate `G` by VCGEN and `C` by COMPILER used to compile the GIP software before deployment.

Related Work. Kelly [5] provides more information on assurance cases and GSN. Weaver [11] documents the use of assurance cases (and case patterns) in software. Assurance cases have been used to address system safety [6], and to justify safety and dependability claims [7]. Arney et al. have developed a set of requirements and a hazard analysis for a generic infusion pump [1]. Goodenough and Weinstock [4] explore demonstrating the quality of the evidence in an assurance case, and using assurance cases for medical devices [12]. Basir et al. [2] have looked at automatically generating safety cases from the formal annotations used to construct Hoare-style proofs of program correctness. Our approach is less automated, but potentially applicable to a wider class of proof-generation techniques. PCC [9] was introduced by Necula and Lee and provides an effective means for providing objective evidence of memory safety properties of low-level. CMC [8] aims to generate proof-certificates by extending model checking algorithms. Chaki et al. [3] have explored combinations of PCC and CMC to generate proof-certificates of expressive properties on low-level programs. Our work is aimed at extending these, and other, formal techniques to provide objective confidence about the safe execution of realistic systems.

Conclusion and Future Work. We report on preliminary work in using assurance cases to bridge the gap between a proof about a program's semantics in a formal system, and its actual behavior in the real world. To this end, we present an assurance case pattern for arguing that a proof is free from various validity hazards. We also instantiate this pattern for a specific application of formal certification technology to an infusion pump software. An important question is if our pattern is instantiable with formal certification schemes other than PCCCMC, and how to make it more robust and complete.

References

[1] D. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky. Generic Infusion Pump Hazard Analysis and Safety Requirements. Technical report MS-CIS-08-31, University of Pennsylvania, October 2008.

[2] N. Basir, E. Denney, and B. Fischer. Constructing a safety case for automatically generated code from formal program verification information. In *Proc. of SAFECOMP*, 2008.

[3] S. Chaki, J. Ivers, P. Lee, K. Wallnau, and N. Zeilberger. Model-driven construction of certified binaries. In *Proc. of MODELS*, 2007.

[4] J. Goodenough and C. Weinstock. Hazards to Evidence: Demonstrating the Quality of Evidence in an Assurance Case. Technical Report CMU/SEI-2008-TN-016, SEI, 2008. in preparation.

[5] T. Kelly. *Arguing Safety*. PhD thesis, Univ. of York, 1998.

[6] T. Kelly and R. Weaver. The Goal Structuring Notation – A Safety Argument Notation. In *Proc. of the Dependable Systems and Networks Workshop on Assurance Cases*, 2004.

[7] L. Millett. Software for Dependable Systems: Sufficient Evidence?, 2007. http://www.nap.edu/catalog.php?record_id=11923.

[8] K. S. Namjoshi. Certifying Model Checkers. In *Proc. of CAV*, 2001.

[9] G. C. Necula. Proof-Carrying Code. In *Proc. of POPL*, 1997.

[10] Stanford Encyclopedia of Philosophy: Defeasible Reasoning, 2005.

[11] R. Weaver. *The Safety of Software – Constructing and Assuring Arguments*. PhD thesis, Univ. of York, 2003.

[12] C. Weinstock and J. Goodenough. Towards Assurance Cases for Medical Devices. Technical Report CMU/SEI-2009-TN-018, SEI, 2009. in preparation.

Towards a Certified Lightweight Array Bound Checker for Java Bytecode

David Pichardie
INRIA
Rennes, Bretagne, France
david.pichardie@irisa.fr

Abstract

Dynamic array bound checks are crucial elements for the security of a Java Virtual Machines. These dynamic checks are however expensive and several static analysis techniques have been proposed to eliminate explicit bounds checks. Such analyses require advanced numerical and symbolic manipulations that 1) penalize bytecode loading or dynamic compilation, 2) complexify the trusted computing base. Following the Foundational Proof Carrying Code methodology, our goal is to provide a lightweight bytecode verifier for eliminating array bound checks that is both efficient and trustable. In this work, we define a generic relational program analysis for an imperative, stack-oriented byte code language with procedures, arrays and global variables and instantiate it with a relational abstract domain as polyhedra. The analysis has automatic inference of loop invariants and method pre-/post-conditions, and efficient checking of analysis results by a simple checker. Invariants, which can be large, can be specialized for proving a safety policy using an automatic pruning technique which reduces their size. The result of the analysis can be checked efficiently by annotating the program with parts of the invariant together with certificates of polyhedral inclusions. The resulting checker is sufficiently simple to be entirely certified within the Coq proof assistant for a simple fragment of the Java bytecode language. During the talk, we will also report on our ongoing effort to scale this approach for the full sequential JVM.

E. Denney, T. Jensen (eds.); The 3rd International Workshop on Proof Carrying Code and Software Certification, pp. 29-29

Towards PCC for Concurrent and Distributed Systems (Work in Progress)

Anders Starcke Henriksen Andrzej Filinski
Department of Computer Science
University of Copenhagen, Denmark
{starcke, andrzej}@diku.dk

Abstract

We outline some conceptual challenges in extending the PCC paradigm to a concurrent and distributed setting, and sketch a generalized notion of module correctness based on viewing communication contracts as economic games. The model supports compositional reasoning about modular systems and is meant to apply not only to certification of executable code, but also of organizational workflows.

1 Introduction

The notion of proof-carrying code is relatively well understood for sequential programs, though substantial technical challenges evidently remain in constructing practically useful PCC systems for realistic application domains. It would clearly be desirable to extend these techniques and results to also allow certification of components of concurrent and distributed systems, but here it is not so clear what a formal certification should even mean in principle, let alone how to realize it in practice.

The immediate difference from a sequential setting is that components are typically running on separate systems, communicating only by exchanging messages. Thus, instead of Hoare-style pre- and post-conditions for a piece of code, we must in general specify its behavior by *communication contracts*: descriptions of which messages may or must be sent between components, depending on the entire previous communication history between them (or some more compact representation thereof). While this change obviously imposes some additional technical difficulties, it does not yet significantly strain the basic notion of certified correctness.

Rather, we consider the main problem with extending PCC to a truly distributed (both physically and administratively) setting to be that even certified components or modules may ultimately depend on external services, whose internal structure will never be available for inspection or certification. Hence, correctness guarantees will in general not be absolute, but only contingent upon correct behavior of some uncertifiable components. But unlike the traditional setting, the client of the certified module will in general not be in a position to assume responsibility for correctness of all submodules used by that module, because those may be provided by third parties, selected by the module implementor, and in principle unknown to the original client.

To reason about such systems in a compositional way, we propose a refined, quantitative model of contract conformance, in which component implementors are rewarded by each other for behavior in accordance with the agreed-upon contract, and penalized for deviations. A properly implemented (i.e., "correct") component is then one that ensures that any fines it may incur for incorrect behavior will be at least matched by the fines it can collect from any faulty subcomponents it depended on. In particular, such a component will never implement a high-assurance service (i.e., one with a high penalty for failure) by relying on a low-assurance one.

Incidentally, such a view of correctness may be relevant even for non-distributed systems, in that a monetary representation of safety or security warranties is inherently more robust in the real world than the mathematically ideal notion of absolute correctness that pure PCC in principle promises. In

E. Denney, T. Jensen (eds.); The 3rd International Workshop on Proof Carrying Code and Software Certification, pp. 30-32

particular, this "putting your money where your mouth is" notion of contract conformance allows seamless integration of modules that have been machine-verified, but are not equipped with independently checkable certificates.

2 A game-theoretic model

To concretize this notion of correctness, we have developed an abstract, game-theoretic model of specification conformance for communicating modules. Here, communication actions are seen as moves, contracts are interpreted as game rules (including any applicable rewards or fines associated with particular moves), and implementations, or processes, correspond to strategies [1]. In the model, we consider both games and strategies at a very abstract level, namely as infinite-state automata; in practical realizations, both contracts and processes would be expressed in suitable languages with more internal structure, to help guide reasoning about conformance.

In general, a module will play simultaneous games with multiple peers, formalized as a *contract portfolio*; a certificate represents a proof that the overall strategy is fiscally sound, i.e., that the cumulative payoff from all the module's games remains non-negative at all times. In particular, a correct module will never be the *first* to break a communication contract, but it may be forced to do so by the previous failure of another module. Semantic correctness, i.e., that a process p satisfies communication contracts $c_1, ..., c_n$ is captured by a coinductively defined relation $\models p : c_1, \ldots, c_n$.

Moreover, the model allows compositional reasoning about correctness, in the sense that two modules may cooperate (by establishing an internal communication contract between them) to implement an external specification. For a contract c, let $\overline{c}$ represent the same contract, but with the roles of the two players interchanged. Then if $\models p_1 : c_1, \ldots, c_n, c'_1, \ldots, c'_{m_1}$ and $\models p_2 : \overline{c_1}, \ldots, \overline{c_n}, c''_1, \ldots, c''_{m_2}$, where only the internal contracts $c_1, ..., c_n$ mention any internal communication links between processes p_1 and p_2, the concurrent composition $p_1 \| p_2$ (straightforwardly definable) will satisfy $\models p_1 \| p_2 : c'_1, \ldots, c'_{m_1}, c''_1, \ldots, c''_{m_2}$. This can be seen as a generalization of the sequential composition rule of Hoare logic, where commands c_1 and c_2 satisfying $\{A\}c_1\{C\}$ and $\{C\}c_2\{B\}$ can be composed into $c_1; c_2$ satisfying $\{A\}c_1; c_2\{B\}$, with no mention of the intermediate assertion C.

3 Towards certification

At this stage, we have only started looking at how to formally represent proofs that a module implementation is formally correct with respect to its contract portfolio, i.e., a sound proof system $\vdash p : c_1, \ldots, c_n$ for semantic correctness. However, once the novel notion of correctness is taken into account, the property we are certifying is ultimately still a traditional safety property; in particular, any failures to satisfy a contract portfolio will be manifestly observable in finite time. In other words, we do not consider unquantified liveness properties such as fairness, or even absence of deadlock; rather, it is only the failure to send a required message by a specific deadline that constitutes an actionable breach of contract. Thus, in principle, standard code-certification techniques should apply without requiring major modifications.

It may be worth considering the significance of code certification in a software-as-service model in the first place: if warranties are ultimately performance-based rather than absolute, do we really need certificates at all? We do, of course, in essentially the same instances as for sequential settings, namely for code not executed by its original creator. That is, in some situation a client (interpreted broadly) wants to not only engage in a game with a contractor, but to actually purchase (or otherwise obtain) that contractor's full strategy, meaning the actual executable code, together the the relevant service agreements with any subcontractors, who now become responsible to the original client directly. In this case,

the client will generally want an absolute correctness proof for the purchased strategy, which can be integrated modularly with that of the client's other games.

4 Context and perspectives

This work was developed in the context of the project *TrustCare: Trustworthy Pervasive Healthcare Services* (`www.trustcare.eu`). Part of the broader goals of the correctness model was that the notion of processes and contracts should not be limited to executable computer code and communication, but be able to serve as a general model of interacting actors (both human and organizational), with internal *workflow* procedures for achieving specific objectives, subject to external constraints. The prototypical scenario is the organization of tasks in a hospital, with the responsible actors representing patients and staff members (doctors, nurses, orderlies, receptionists, pharmacists, etc.), interacting not only by exchange of information, but also by physical actions, such as medical tests and procedures.

In general, the interdependencies between these actions may be quite complex: individual patients may suffer from multiple conditions, drugs and treatments may interact in complex ways, test results may be time sensitive, etc. Thus, developing and certifying a set of workflows and best-practice guidelines for individual staff members, to ensure that all patients are treated safely and efficiently, is a non-trivial task, well suited for (at least partial) automation. The quantitative model also allows the representation of relative importance of potentially conflicting constraints; for instance, (combinations of) actions that would significantly endanger a patient's life would be assigned higher negative payoffs than those which are merely wasteful of resources; and best-practice workflows will ensure that robust checks and balances are in place that help avoid potentially dangerous outcomes resulting from isolated minor errors by individual actors.

One might expect that, as patient records become increasingly electronic, formal certification of large-scale institutional workflows would become a licensing requirement for health care providers, in line with basic hygiene and staff training requirements. However, at the present time, we are targeting the model primarily towards certification of traditional code.

Acknowledgment The authors gratefully acknowledge the substantial contributions of Tom Hvitved of the *3gERP (Third-generation Enterprise Resource Planning)* project to the development of the contract model.

References

[1] Anders Starcke Henriksen, Tom Hvitved, and Andrzej Filinski. A game-theoretic model for distributed programming by contract. In *Workshop on Games, Business Processes, and Models of Interaction*, Lübeck, September 2009. Gesellschaft für Informatik. To appear.

Proof compression and the Mobius PCC architecture for embedded devices

Thomas Jensen
CNRS
IRISA, Campus de Beaulieu, F-35042 Rennes, France

The EU Mobius project[1] has been concerned with the security of Java applications, and of mobile devices such as smart phones that execute such applications. In this talk, I'll give a brief overview of the results obtained on on-device checking of various security-related program properties. I'll then describe in more detail how the concept of certified abstract interpretation and abstraction-carrying code can be applied to polyhedral-based analysis of Java byte code in order to verify properties pertaining to the usage of resources of a down-loaded application. Particular emphasis has been on finding ways of reducing the size of the certificates that accompany a piece of code.

Such analyses will produce program invariants in the form of fixpoints of abstract transfer functions and the present work has been concerned with several aspects of how to simplify and reduce the size of these fixpoints:

- Abstract interpretations will often produce more information that necessary for proving a particular program property. I will introduce the notion of a **witness** of a property together with a technique for **pruning** program invariants to provide the minimum information necessary to prove a particular property.
- In the context of polyhedral-based abstract interpretation, an essential part of verifying a proposed invariant is the checking of polyhedral inclusions. I'll describe a notion of certificate for checking polyhedral inclusions based on a use of Farkas' lemma that reduces inclusion checking to a few simple matrix computations.
- The checking of a proposed invariant involves a certain amount of re-computing the invariant. It is therefore possible to reduce the amount of information that is being transmitted in the certificate because the checker will reconstruct it anyway. This leads to general **fixpoint reconstruction** algorithms that generalize the dedicated algorithms from lightweight bytecode verification.

Checkers of such compressed certificates have been developed in the proof assistant Coq and extracted to be executed on several types of mobile devices.

References

Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

Frédéric Besson, Thomas Jensen, and David Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 364(3):273–291, 2006.

Frédéric Besson, Thomas Jensen, Tiphaine Turpin. Small witnesses for abstract interpretation based proofs. In *Proceedings of the 16th European Symp. on Programming (ESOP 2007)*, Springer LNCS vol. 4421, 2007.

E. Denney, T. Jensen (eds.); The 3rd International Workshop on Proof Carrying Code and Software Certification, pp. 33-33

[1]This work was partialy supported by the EU FET project FP6-015905 Mobius