riacs Research Institute for Advanced Computer Science

Synthesizing Safety Conditions for Code Certification Using Meta-Level Programming

Jutta Eusterbrock

RIACS Technical Report 03.17

Synthesizing Safety Conditions for Code Certification Using Meta-Level Programming[1]

RIACS Technical Report 03.17

[1] This work was supported in part by QSS Inc.

Synthesizing Safety Conditions for Code Certification Using Meta-level Programming

Jutta Eusterbrock
QSS/ NASA Ames Research Center, CA 94035
JEusterbrock@acm.org

Abstract

In code certification the code consumer publishes a safety policy and the code producer generates a proof that the produced code is in compliance with the published safety policy. In this paper, a novel viewpoint approach towards an implementational re-use oriented framework for code certification is taken. It adopts ingredients from Necula's approach for proof-carrying code, but in this work safety properties can be analyzed on a higher code level than assembly language instructions. It consists of three parts: (1) The specification language is extended to include generic pre-conditions that shall ensure safety at all states that can be reached during program execution. Actual safety requirements can be expressed by providing domain-specific definitions for the generic predicates which act as interface to the environment. (2) The Floyd-Hoare inductive assertion method is refined to obtain proof rules that allow the derivation of the proof obligations in terms of the generic safety predicates. (3) A meta-interpreter is designed and experimentally implemented that enables automatic synthesis of proof obligations for submitted programs by applying the modified Floyd-Hoare rules. The proof obligations have two separate conjuncts, one for functional correctness and another for the generic safety obligations. Proof of the generic obligations, having provided the actual safety definitions as context, ensures domain-specific safety of program execution in a particular environment and is simpler than full program verification.

1 Introduction

Code certification provides a mechanism for insuring that a host, or code consumer, can safely run code delivered by a code producer. The host specifies a safety policy as a set of axioms and inference rules. In addition to the program, the code producer delivers a certificate, ie., a formal proof of safety expressed in terms of those rules that can be easily checked. AutoBayes (cf. [FSW02]) addresses the problem of combining automatic code generation from high-level specifications together with code certification. AutoBayes automatically generates imperative programs for data analysis that are annotated with pre-conditions, post-conditions, loop invariants, and with some other annotations from compact high-level specifications. The process of proving functional correctness is different from proving safety of program executions. When analyzing safety, environmental constraints that cannot be anticipated while the program is being developed and the states that can be reached during execution need to be taken into account. The code consumer defines exactly under what conditions execution is considered to be safe.

This paper applies the viewpoint architecture devised by the author (cf. [Eus99]) to enable practical implementations for automated certification of annotated programs. *Viewpoints* (VPs) are independent intermediate software components, introduced for the loosely coupling of multi-paradigm knowledge sources. Generic methods are derived domain-independently with respect to associated generic theories. A VP specifies how domain-specific knowledge sources correspond to a generic theory. Using a VP as application specific context, any associated generic method can then be specialized for the particular domain. The framework adopts ingredients from Necula's approach for proof-carrying code (cf. [Nec97]). A safety policy is considered to consist of the safety rules and the interface, but in this work generic safety properties are analyzed on a higher code level rather than assembly language instructions. A three-stage approach is taken. Firstly, in this paper, the safety interface to the environment is analyzed in terms of generic safety pre-conditions. Application-specific safety requirements are expressed by providing them as instantiations of the generic safety predicates. Secondly, the Floyd-Hoare inductive assertion method is extended and modified to yield a set of verification rules that especially state for all operations the associated generic safety pre-conditions. Thirdly, a meta-interpreter is designed and implemented that synthesizes proof obligations for annotated programs by applying the modified proof rules. The obligations have two separate conjuncts, one for functional correctness and another for the generic safety conditions.

Having decoupled the definition of the safety interface from the safety rules provides a flexible re-use oriented framework for the analysis of various domain-specific safety policies. It is exemplified analyzing a basic but nontrivial level of safety, including array bounds, undefined values, and finite domain constraints in a way that is simple, efficient, and - most important - easy to plug into the existing AutoBayes architecture. When the proof obligations are generated, the user can use all the power of the underlying simplifier and theorem prover to resolve the correctness obligations. Once the correctness proofs are successfully done, it is guaranteed that the synthesized programs always satisfy the functional correctness requirements. Correctness proofs can be re-used in a variety of environments. In addition, a proof of the generic obligations, having provided the actual safety definitions as context, ensures domain-specific safety of program execution in a particular environment. These proofs can be automatically performed using highly specialized tools like constraint solver or type checker and, therefore, the whole process is simpler than full program verification.

The remainder of this paper is organized as follows. Section 2 gives an overview of the certification framework. Section 3 exemplifies the concept of safety interfaces. Section 5 presents the design of the meta-interpreter that enables the automatic synthesis of generic safety conditions and demonstrates it on examples. Finally, this framework is compared with related work.

2 Overview of the Certification Framework

Proof-carrying code is a framework for verifying the safety of machine-language programs with a machine-checkable proof. Following Necula (cf. [Nec97]) the idea is that the code consumer defines and publishes a *safety policy*. The code producer constructs in the stage named *code certification* a formal proof that a program respects the published safety policy. First it computes its safety predicate using verification-condition generation rules. The safety predicate is a predicate in first-order logic with the property that its validity is a sufficient condition for ensuring compliance with the safety policy. The code consumer downloads both the code and the proof. Then, it checks whether the safety proof is a valid proof of the safety predicate. The code consumer executes the machine-code without performing additional run-time

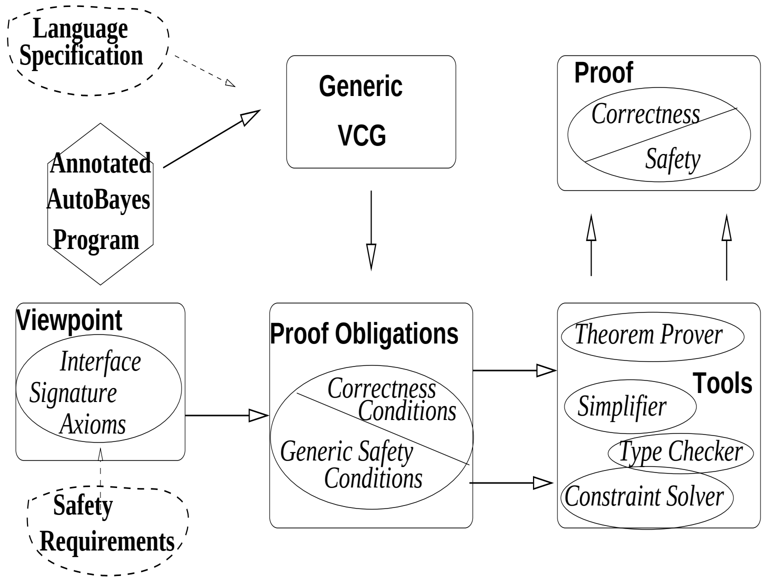

Figure 1: Overview of the Certification Framework

checks.

This paper presents a generic framework for verification condition generation that doesn't depend upon particular safety requirements. For its realization, the viewpoint concept which was devised by the author to facilitiate the design of re-usable generic methods is being applied. Adopting Necula's ([Nec97]) approach, a safety policy consists of two main components, the safety interface and the safety rules. The safety interface specifies the requirements for the execution of program steps in interaction with the environment of interest to be secure. While Necula's verification condition generator embodies a specific abstract machine model reflecting the crucial safety parameters of the anticipated execution environment, the proposed approach generalizes the process by employing additional generic safety predicates in the verification condition generator. These virtual generic safety pre-conditions ensure that the state after operator application and the state before is safe. They are computed and propagated during verification condition generation, however no run-time checks are required. The generic safety predicates can be instantiated providing the safety interface determining the concrete situational safety requirements of the executing environment as a viewpoint. Based upon these extended verification rules, a meta-interpreter is devised that enables the computation of proof obligations for submitted annotated programs. The obligations have two separate conjuncts, one for functional correctness and another for the generic safety obligations. Automated theorem provers can be used for verifying functional correctness. However, first-order theorem provers are not well suited for type checking, algebraic simplifications, numeric and symbolic arithmetic calculations which naturally arise from considerations about safety of imperative programs. In most cases, the particular safety obligations can be easily discharged by specialised and efficient analysis tools. Figure 1 depicts the resulting architecture.

2.1 Specification and Example Scenario

AutoBayes synthesizes programs which are imperative programs constructed of declarations and assignments, conditionals, loops, iterations, sequences and block statements that operate on multi-dimensional arrays. Programs are annotated with logical assertions in a Floyd-Hoare logic style, such as pre-conditions, post-conditions and loop invariants. Pre-conditions specify the valid values of the input parameters. Post-conditions specify the output and the (possibly modified) environment. Invariants state properties that are true at each iteration in a loop. Annotations support design and verification of software units. The code is in the form of an intermediate language. The specification terms are Boolean expressions with the addition of quantifiers, arbitrary user-defined predicates and basic types. The syntax of the specification language is summarized in Figure 2.

To illustrate the proposed generic framework for safety certification, the following demonstration scenario is considered. The program in Figure 3 is encoded using a pseudo language. It is intended to abstract a Javascript program which sequentially selects an array of images according to their width and height and encodes the selection as HTML code, which in turn causes the placement of these pictures into a display area while being interpreted by a browser. The actual display depends on the execution environment as the computer platform, screen resolution or the used browser. In one environment, the generated display may look as it was designed to be, taking another environment, the image display created with the same program can be completely different or sometimes the browser doesn't show anything at all. An execution environment is considered to be unsafe, if at some stage during interpretation images or parts of them disappear. The execution can be re-

Var	$x, x_1, \ldots, x_r$
$Expr$	$expr ::= x \vert e_1 + e_2 + \ldots + e_n \vert e_1 - e_2 \vert e_1 * e_2 \vert e_1 \ e_2 \vert substitute(x, e) \vert$ $assign(a, e_1, e_2) \vert access(a, e) \vert error,$ where e_i is of type $Expr$
$Types$	$\tau : int \vert bool \vert array(\tau, e) \vert array(\tau, e_1, e_2)$
$Predicate$	$true \vert false \vert P_1 \wedge P_2 \vert P_1 \Rightarrow P_2 \vert \forall x : P_x \vert \exists x : P_x \vert$ $e_1 = e_2 \vert e_1 \neg e_2 \vert e_1 \geq e_2 \vert e_1 < e_2 \vert e : \tau \vert$ $safe\_assign(x, e) \vert safe\_expr(e_1) \vert safe\_assign(a, e_2, e_3) \vert$ $safe\_access(a, e_3),$ where P_i is of type $Predicate$
Inv	$\iota ::= ANN\_INV(P, \{x_1, \ldots, x_r\})$
$Spec$	$\sigma ::= Prog : (PRE = P_1, POST = P_2)$

Figure 2: Syntax of the Assertion Language

garded as safe, if the subsequent layout process yields the intended results. More precisely, a display area is defined as $0 \leq x \leq width, 0 \leq y \leq height$, where *width* and *height* shall be integer constants, taking the upper left corner as global origin. The definition of the size shall be part of the safety specification. However, size measures can be different depending on the current environment. Whereas the size of the display area could be specified using decimal measures, normally, script programs suppose sizes to be stated in pixels. Further on, it is supposed that the behavior of the execution environment is characterized by known system parameters and the actual layout at each stage during execution can be computed as combination of system parameters and values for program variables. However, at the time when the program is being developed, the behavior of the target execution environment is unknown because system parameters are subject to dynamic change. In this demonstration scenario, it is supposed that at all stages during display, a context-dependent relative origin can be determined. Its coordinates are described in relation to the top left corner of the display area. The browser positions the image taking this relative origin as top left corner leaving a margin at the left and upper border of the display area. After having assigned a single image to the display array, the context-dependent relative origin is re-computed.

```
PRE={true}; PROC(place-images)= {
  DECLS{input: array(images,gif,2),
        output: array(d_area,gif,2),
        local:  scalar(i_width,integer),
                scalar(i_height,integer)}
  STMTS{i_width:=150*2;i_height:=350;
          d_area(0):=access(images,2);
          d_area(2):=access(images,1);
          d_area(1):=access(images,0)}};
  POST={exists im: access(d_area,2)=
                assign(images,0,im)}
```

Figure 3: Source Program

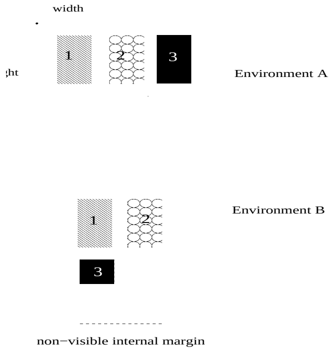

Figure 4: Context-dependent Interpretation

2.2 Safety

To analyze safety of embedded systems, it is necessary to define the legal states and actions, to show that the initial states are safe and to analyze the conditions under which the defined actions transform legal states into legal states. Technically this analysis is done by pre-condition analysis (cf. [Hei96]). [HCEH96] describes a framework for incorporating safety requirements into operational Z specifications of software for embedded systems. Their approach is to produce additional pre-conditions which restrict the applicability of operations. Operations may be applied to a given state only if it leads to a safe successor state. In this work, this technique is translated to analyze safety in addition to functional correctness of program executions. The Hoare semantics is refined by considering the various states program and additional environment variables can take during program execution using a trace semantics (cf. [Cou01]). A program execution is a sequence of states, starting from an initial state for which the pre-condition is true, then moving from one state to the next and resulting in a final state which either satisfies the post-condition or is incorrect, assuming the program execution is terminating. A distinction is made between safe and unsafe states. A program execution is considered to be safe if and only if each computation of program and environment variables at each state during the execution complies with the host-provided domain-specific safety policy. It is supposed that rules of safe programming practice and the safety properties that a program should satisfy can be specified by abstract types. Safety is considered as type correctness of variables and operator applications that must hold in all states during the program execution.

The basic idea of safety checking is then to show that, firstly, the initial states are safe. This requires checks performed by the program that the precondition is true for the initial values and which is outside the scope of this paper. Secondly, the input arguments for operator application are safe with respect to its signature. Thirdly, operator application results in a safe state. In order to achieve a formal safety model, the definition by Necula and Lee (cf. [NL98]) is extended. A safety policy consists of two main components, the

- *Interface description*, which is the boundary between the foreign software code and the code consumer. It contains a description of the formal type parameters for which syntax and semantic safety constraints can be given.
- *Safety rules*, which describe all authorized operations and their associated safety pre-conditions.

The generalized safety policy consists of a *viewpoint* and *generic safety rules*. The following subsections discuss these two components in more detail.

2.3 Viewpoints as Extension of Interface Descriptions

It is widely accepted that precise interface descriptions make it easier to develop complex software systems from software components by controlling and structuring the dependencies between them. The interface is first specified, then the software which satisfies or uses the interface description is implemented. In this framework, the meaning is slightly different. An interface is an abstract meta-level description that specifies the safety requirements that should hold for the interactions between the foreign code and its environment. During the certification process, it is formally verified that the given code conforms to its interface description. A safety interface is defined by a list of variable and type declarations, ie. the signature, against which the code will be certified. Whenever a type is declared, it can be attached safety axioms in the form of algebraic expressions or constraints. Type declarations define the states to be safe and the data-representation axioms to be preserved. The interface may include descriptions which variables are initialized in the beginning.

In this paper, the viewpoint framework (cf. [Eus99]) is being applied to make domain-specific safety requirements accessible for generic verification. Generally, the starting point are generic predicates and their logical axiomatization. Generic methods are derived in terms of the generic predicates. A generic method is then specialised to the environment of interest by providing the viewpoint as context. A viewpoint specifies how an abstract generic theory relates to a particular domain. It consists of rules which provide domain-specific definitions for the generic predicates such that the abstract properties remain valid. Analyzing safety, generic requirements are stated in terms of the generic safety predicates $safe\_expr$, $safe\_assign$. $safe\_expr(expr)$ is a generic safety condition with the intended meaning that for all admissible inputs the evaluation of $expr$ doesn't cause an error. The intended meaning of $safe\_assign$ is, the safety condition is provable only if assignment leads to a safe successor state, provided that the state before assignment is safe. It is up to the actual safety policy to define the meaning of these generic safety predicates. Examples are discussed in Section 3.

2.4 Extending the Pre-Condition for Generic Safety

Hoare (cf. [Hoa69]) introduced the notation $\{Pre\}Q\{Post\}$ for partial correctness where Pre and $Post$ are predicates specifying the pre-conditions and the desired result, respectively, for program Q. That is, if the assertion Pre is true before initiation of a program Q, then assertion $Post$ will be true when it completes execution. Hoare presented the necessary axioms and inference rules for reasoning about programs written in a simple imperative language. Using Hoare Logic, a proof of $\{Pre\}Q\{Post\}$ is a sequence of sentences, the first of which is $\{Pre\}Q\{Post\}$, and each sentence is either a Lemma to be proven in the underlying logical system or a simpler sentence of the form $\{Pre'\}Q'\{Post'\}$. Each sentence in the sequence is derived from a previous line by applying a proof rule. The difficulty with this approach is that it is not clear how one determines the Lemmas to automate the process.

This problem is solved using Floyd-style verification condition generators (VCGs), such as cf. [ILL75]. To build a VCG the axiomatic Hoare rules are reformulated to produce a deterministic set of rules that generate subgoals and ultimately verification conditions. The standard approach of verifying loops using Floyd-style VCGs involves introducing loop invariants explicitly, which is a challenge for any theorem-proving technique and often requires user intervention. Programs automatically synthesized by AutoBayes are annotated with loop invariants and a standard VCG allows to automate the process of producing verification conditions for proving programs correct with respect to their specifications. These verification conditions can then be resolved, dependent on the underlying logic, possibly with the aid of an algebraic simplifier or theorem prover. Functional correctness doesn't necessarily include safety of program execution. For example, the annotated example program in Figure 3 could be proven correct by applying the Hoare verification rules. However, program execution in a particular environment can be unsafe. In this simplified example, the faulty verification can be avoided by further assertions that attach domain-specific safety properties to single program statements. Generally, the hard encoding of safety obligations is a very tedious task. Moreover, such an approach is very unflexible. Each time when the safety requirements change, safety annotations need to be re-generated from scratch.

In this paper, the standard Floyd-style rules for VCG are extended to cover safety of program executions. A meta-interpreter then allows to automate the process of producing proof obligations whose validity ensures programs functional correct and safe to execute with respect to their specifications. All security-related operations the program can perform at all stages during program execution are considered. When the VCGen sees a safety-related operation that changes the state of variables, a generic safety condition is launched and used to strengthen the pre-condition.

3 Safety Viewpoints

The basic idea is to identify rules of safe programming practice concerning safety properties on various levels (hardware, runtime, applications), formalize them, and making these descriptions accessible for a generic approach towards safety verification through a safety viewpoint Γ_{safe}, consisting of the signature, axioms and lifting rules which provide domain-specific definitions for the generic safety predicates.

3.1 Type Declaration/Signature

Many software bugs come from small mistakes. Programmer sometimes forget to initialize a variable or omit an argument to a function. If a loop misses the incrementation step, it will loop forever. Invalid arithmetic operations, e.g., division by zero, may yield results which are not defined or in the wrong order of magnitude. In some cases, erroneous programs are actually dangerous. The flight of the Ariane 5 space launcher ended in an explosion due to an input conversion function which could not cope with the large numbers (cf. [Nus97]). Some common programming errors, whose formalization will be sketched are

- accessing a non-initialized array field;
- accessing an element from an array with an index outside its bounds;
- overflow / underflow of arithmetic operations.

Some bugs can be avoided by switching to a safer language. In a perfect strongly typed language, the type carries all the information about the safe states and verification of type safety is done by the compiler. None of the strongly typed languages are considered perfect. Moreover, safer programming languages cannot prevent many other security bugs, especially those involving higher level semantics like the environment specific non-visible display boundaries in Figure 4. The intention of this work is to provide a framework that allows to express and check domain-specific safety requirements in addition to common runtime errors, especially those that can be expressed by finite domain constraints.

The safety signature is an abstract description of the formal parameters for which syntax and semantic safety conditions can be given. It is defined by a list of type declarations for variables and requirements on calling conventions (like input-output arguments) to be obeyed by foreign applications.

Firstly, the types of the safety specification shall be a subset of the program declaration. Program declarations are viewed as generic type declarations for which the actual safety interface description provides domain-specific instantiations. This requires a signature mapping from the program declaration to the actual signature of the safety specification. Consider for example the program declaration

```
DECLS{input: array(images,gif,2),
      output: array(d_area,gif,2),
      local: scalar(i_width,integer),
             scalar(i_height,integer)}
```

This could be translated into the safety signature

$$\begin{aligned} VARS \quad & images : array(gif, *, 2) \wedge d\_area : array(gif, 2) \wedge \\ & i\_width : integer \wedge i\_height : integer. \end{aligned} \tag{1}$$

The additional pre-condition below states that input values need to be initialized prior to program execution.

$$PRE \quad \exists x_{init}[i] : gif \wedge access(images, i) = x_{init} \wedge 0 \leq i \leq 2. \tag{2}$$

Secondly, there can be multiple possible signatures for the program code, each being a useful abstraction for validating particular safety requirements. The safety signature can augment the type specifications entailed in program interfaces. This could be necessary, for example, to analyze performance properties or consumption of resources. These properties can be security-relevant, however are typically not considered when proving functional correctness. In the example discussed here, an additional context variable that maintains the dynamic coordinates of the "relative origin" for placing images is needed as part of the safety assertions, ie.,

$$POST = \exists x : access(rel\_origin, 0) = x \wedge \exists y : access(rel\_origin, 1) = y \tag{3}$$

3.2 Safety Axioms

A safety policy has to provide the axioms that hold taking into account the particular safety requirements. Safety axioms are a formal abstract description of the data-representation axioms to be preserved in the specific environment such that the execution is considered to be safe.

Overflow/Underflow As Hoare points out in [Hoa69], infinite arithmetics satisfies $\neg\exists x \forall y (y \leq x)$ where all finite arithmetics satisfy: $\forall x (x \leq max)$ and "max" denotes the largest integer represented. Furthermore, Hoare distinguishes the three treatments of overflow

$$\begin{aligned} \neg\exists x(x = max + 1) & \quad \text{(strict interpretation)} \\ max + 1 = max & \quad \text{(firm boundary)} \\ max + 1 = 0 & \quad \text{(modulo arithmetics)} \end{aligned}$$

For example, the mathematical set of integers can be implemented in JAVA by the built-in integer data types `byte`, `short`, `int`, `long` which all have a different, but finite range (cf. [BS02]). The actual behavior could also be based upon the computer architecture. In the remainder of this paper, it is supposed that the finite domain of integer values is constrained by a maximal and minimal bound, e.g.:

$$i : integer \Leftarrow i \geq minint \wedge i \leq maxint, minint = -2^5, maxint = 2^5 \quad (4)$$

Array Access (Multi-dimensional arrays) are axiomatized as an abstract data type $array(\tau, e_1)$, $array(\tau, e_1, e_2)$, ..., $array(\tau, e_1, e_2, \ldots, e_n)$, $n \in \mathbb{N}$, where τ denotes the type of the array elements and $e_1, e_2, \ldots, e_n$ its size. Arrays used in this paper are assumed to start at index 0, will be assigned values by the operation *assign* and can be accessed by the operation *access*. In the one-dimensional case, access and assignment on arrays satisfy the following axioms (cf. [LS79]):

$$\begin{aligned} &access(assign(a, j, expr), i) = access(a, i) \Leftarrow i \neq j. \\ &access(a, i) = expr \Leftarrow i = j \wedge assign(a, j, expr). \\ &access(a, j) = error. \end{aligned} \quad (5)$$

Side Effects The coordinates of the relative origin change dynamically as result of assignments to the display area. There cannot be any side-effects in assertions. If *Post* denotes a state then $select(Post, rel\_origin)$ denotes the contents of rel_origin and $update(select(Post, rel\_origin), assign(d\_area,, expr))$ denotes the state obtained from rel_origin by assignment of *expression* to a field in d_area.

3.3 Lifting Rules

Regarding a basic imperative programming language, the semantic change of state is caused by the assignment operator and arithmetic operations. A generic verification condition generator is devised using the meta-predicates $safe\_assign$, $safe\_expr$ to mark context-dependent verification conditions. $safe\_expr(expr)$ denotes the assertion that the evaluation of the symbolic expression *expr* must be safe, ie., according to the context-dependent typing axioms, the evaluation doesn't cause an error for any admissible input. The generic safety condition $safe\_assign(lhs, rhs)$ is used to augment the original pre-condition of each assignment $lhs := rhs$ with the general meaning that the state after assignment is legal, provided that the statement before is legal and taking into account side-effects. Domain-specific safety requirements can be introduced by attaching finite domain constraints to the typed variables available in the program. Analyzing the scenario stated in Figure 3, for example, the absolute sizes of the domain-specific display area need to be taken into account for safety analysis. For example, the maximal allowable display area on specific laptops embraces 1024x768 pixel and the initial coordinates of a virtual origin could denote the supposed margin from the upper left corner. Applying the viewpoint concept (cf. [Eus99]), lifting rules are used to associate domain-specific requirements with generic problem solving steps. Figure 5 establishes the corresponding definitions.

$$
\begin{array}{ll}
\mathit{VARS} & \mathit{images} : \mathit{array}(\mathit{gif}, 2) \wedge \mathit{d\_area} : \mathit{array}(\mathit{gif}, 2) \wedge \\
 & \mathit{i\_width} : \mathit{image\_width} \wedge \mathit{i\_height} : \mathit{image\_height} \wedge \\
 & \mathit{d\_h} : \mathit{display\_height} \wedge \mathit{d\_w} : \mathit{display\_weight} \wedge \\
 & \mathit{rel\_origin} : \mathit{array}(\mathit{integer}, 1). \\
\mathit{INITIAL} & \mathit{assign}(\mathit{rel\_origin}, 0, 50) \wedge \mathit{assign}(\mathit{rel\_origin}, 1, 50). \\
\mathit{CONSTRAINTS} & \mathit{image\_width}(x) \Leftarrow x : \mathit{integer} \wedge x \geq 0 \wedge x \leq 350. \\
 & x : \mathit{image\_height} \Leftarrow x : \mathit{integer} \wedge x \geq 0 \wedge x \leq 350. \\
 & x : \mathit{display\_width} \Leftarrow x : \mathit{integer} \wedge 0 \leq x \leq 768. \\
 & x : \mathit{display\_height} \Leftarrow x : \mathit{integer} \wedge 0 \leq x \leq 1024. \\
 & i : \mathit{integer} \Leftarrow i \geq -2^{10} \wedge i \leq 2^{10}. \\
\mathit{UPDATE} & \mathit{update}(\mathit{select}(\mathit{Post}, \mathit{rel\_origin}), \mathit{assign}(\mathit{d\_width}, \_, \_)) = \\
 & \quad [\mathit{assign}(\mathit{rel\_origin}, 0, \mathit{access}(\mathit{rel\_origin}, 0) + \mathit{i\_width}), \\
 & \quad \mathit{assign}(\mathit{rel\_origin}, 1, \mathit{access}(\mathit{rel\_origin}, 1) + \mathit{i\_height})]. \\
\mathit{LIFTING} & \mathit{safe\_assign}(\mathit{i\_width}, \mathit{expr}) \Leftarrow \mathit{safe\_expr}(\mathit{expr}) \wedge \\
 & \qquad\qquad\qquad\qquad \mathit{expr} : \mathit{image\_width}. \\
 & \mathit{safe\_assign}(\mathit{i\_height}, \mathit{expr}) \Leftarrow \mathit{safe\_expr}(\mathit{expr}) \wedge \\
 & \qquad\qquad\qquad\qquad \mathit{expr} : \mathit{image\_height}. \\
 & \mathit{safe\_assign}(\mathit{d\_area}, i, \mathit{image}, [\mathit{exp}0, \mathit{exp}1]) \Leftarrow 0 \leq i \leq 1 \wedge \\
 & \quad \mathit{exp}0 : \mathit{display\_width} \wedge \mathit{exp}1 : \mathit{display\_height}. \\
 & \mathit{safe\_expr}(\mathit{expr}) \Leftarrow \neg(\mathit{expr} = \mathit{error}).
\end{array}
$$

Figure 5: Context-dependent Definitions for Generic Safety-predicates

4 Generic Safety Rules

In [Fra96] the term verification conditions and the relation

$$Post \vdash Q \rightarrow Pre, VCs$$

is introduced with the following meaning: the post-condition *Post* is true after a terminating execution of the program Q, if the pre-condition *Pre* is true before executing the program, and if the set of verification conditions VCs contains only valid formulas. Based upon the Floyd-Hoare inductive assertion method, proof rules for the generation of verification conditions VCs are described.

In this work, in addition to verification of functional correctness, the goal is to show that the execution of synthesized code is safe. A safety policy is specified by the viewpoint Γ_{safe} and a set of safety rules. To achieve safety rules that allow the automatic computation of safety conditions, the above cited framework (cf. [Fra96]) for verification condition generation is extended. The relation

$$Post \vdash Q \rightarrow Pre \wedge SafeExpr, VCs :: \Gamma_{safe}$$

is introduced with the following meaning: the post-condition *Post* is true after a terminating execution of the program Q, if the pre-condition *Pre* is true before executing the program, and if the set of verification conditions VCs contains only valid formulas. *SafeExpr* entails all values of program variables at intermediate stages during program execution are allowable, provided that they comply with the requirements in Γ_{safe}.

The resulting modified proof rules to compute the weakest pre-condition, the safety expression and verification conditions are summarized in the subsections 4.1-4.3. The VCGen does not depend on a particular safety theory Γ_{safe}. Γ_{safe} is a domain-specific safety interface as exemplified in the previous section and provided by the code consumer. Subsection 4.4 states the rule to verify safety and functional correctness.

4.1 Assignment

Let $P[x/expr]$ denote the assertion that is like P except that *expr* has been substituted for the occurrences of x.

Variable Assignment

$$\begin{aligned} Post \vdash x := expr \rightarrow Post[x/expr] \wedge \\ safe\_assign(x, expr)[] :: \Gamma_{safe} \end{aligned} \tag{6}$$

Array Assignment

$$\begin{aligned} Post \vdash a[index] := expr \rightarrow \{Post[access(a, index)/expr]) \wedge \\ \wedge safe\_assign(a, index, expr)\}, [] :: \Gamma_{safe} \end{aligned} \tag{7}$$

4.2 Control Structures

Skip

$$Post \vdash skip \rightarrow Post, [] :: \Gamma_{safe} \tag{8}$$

Sequencing

$$\frac{\begin{array}{c} Post \vdash STMT_2 \rightarrow Pre_2, Conds_2 :: \Gamma_{safe} \\ Pre_2 \vdash STMT_1 Pre, Conds_1 :: \Gamma_{safe} \end{array}}{Post \vdash STMT_1; STMT_2 \rightarrow Pre, Conds_1 \cup Conds_2 :: \Gamma_{safe}} \tag{9}$$

Conditional Statement

$$\frac{\begin{array}{c} Post \vdash STMT_1 \rightarrow Pre_1, Conds_1 :: \Gamma_{safe} \\ Post \vdash STMT_2 \rightarrow Pre_2, Conds_2 :: \Gamma_{safe} \end{array}}{\begin{array}{c} Post \vdash \text{ if } B \text{ then } STMT_1 \text{ else } STMT_2 \rightarrow (B \Rightarrow Pre_1) \wedge (\neg B \Rightarrow Pre_2), \\ Conds_1 \cup Conds_2 :: \Gamma_{safe} \end{array}} \tag{10}$$

While Loop

$$\frac{Inv \vdash STMT \rightarrow Pre, Conds :: \Gamma_{safe}}{\begin{array}{c} Post \vdash \text{ while } B \text{ inv}\{\text{Inv}\} \text{ do } STMT \rightarrow Inv, \\ \{Inv \wedge B \Rightarrow Pre, Inv \wedge \neg B \Rightarrow Post\} \cup Conds :: \Gamma_{safe} \end{array}} \tag{11}$$

Assertions: Rules of Consequence

$$\frac{Post \vdash STMT \rightarrow Pre, Conds :: \Gamma_{safe}}{Post \vdash STMT PreAss \rightarrow PreAss, \{PreAss \Rightarrow Pre\} \cup Conds :: \Gamma_{safe}} \tag{12}$$

$$\frac{PostAss \vdash STMT \rightarrow Pre, Conds :: \Gamma_{safe}}{Post \vdash PostAss STMT \rightarrow Pre, \{PostAss \Rightarrow Post\} \cup Conds :: \Gamma_{safe}} \tag{13}$$

4.3 Extensions

Iteration A way to get at a rule for FOR loops is to unfold the FOR into a loop body followed by a WHILE and then apply the WHILE rule and composition/consequence. This yields

$$\frac{Inv[i+1/i] \vdash STMT \to Pre, Conds :: \Gamma_{safe}}{\begin{array}{c} Post \vdash \text{ for } i := e\_0 \text{ until } e\_j \text{ inv}\{\text{Inv }\} \text{ do } STMT \to Inv[e\_0/i], \\ \{(Inv \wedge (i \leq e\_j)) \Rightarrow Pre), Inv \wedge (i > e\_j) \Rightarrow Post\} \cup Conds :: \Gamma_{safe} \end{array}} \tag{14}$$

Composition of Rules of Consequence

$$\frac{PostAssert \vdash STMT \to Pre, Conds :: \Gamma_{safe}}{\begin{array}{c} Post \vdash PostAssertSTMTPreAssert \to \{PreAssert\}, \\ \{PostAssert \Rightarrow Post, PreAssert \Rightarrow Pre\} \cup Conds :: \Gamma_{safe} \end{array}} \tag{15}$$

4.4 Verifying Consistency and Correctness

$$\frac{\begin{array}{l} Post \vdash STMT \to SafeExpr \wedge WPre, Conds :: \Gamma_{safe} \\ \vdash Conds \cup \{Pre \Rightarrow WPre\} \\ \Gamma_{safe} \vdash \neg(\exists x : \neg SafeExpr(x) \wedge (WPre \cup Conds)) \end{array}}{\{Pre\}Decls; STMT\{Post\} :: \Gamma_{safe}} \tag{16}$$

5 Automatization of Safety Certification

5.1 Synthesis of Generic Safety Conditions

The basic approach in this work to building a software component that automatically extracts generic safety conditions from given programs, is to develop an appropriate meta-interpreter. Having derived the extended condition generation rules (cf. Section 4), the implementation of a meta-interpreter that computes safety conditions is straightforward and sketched in Figure 6, implementing a subset of the verification rules assembled in section 4. In AutoBayes, synthesized programs and its annotations are represented as ground terms. Assertions like pre-conditions, post-conditions, invariants and single program statements can be accessed through selector functions. The predicate *scg* is used to implement single relationships among program specifications and safety conditions, according to the rules composed in Section 4 and operates on the AutoBayes term representation of its objects.

Meta-interpreter

Sorts :	$prog\_term, formula, set\_of\_formula$
Variables :	$WPre, Pre, Post : formula;$
	$VCs : set\_of\_formula; Prog : prog\_term.$
Operators :	$decompose\_prog : prog\_term \rightarrow prog\_term \times prog\_term$
Goals	$\leftarrow scg(+Post, +Prog, +Pre, -VCs)$
	$\leftarrow scg\_i(+Post, +Prog, -WPre, -VCs)$
Axioms	

$$\begin{array}{ll}
scg(Post, Prog, Pre, VCs) \leftarrow scg\_i(Post, Prog, WPre, VCs\_i) & \\
\quad VCs = \{Pre \Rightarrow WPre\} \cup VCs\_i. & \\
scg\_i(Post, skip, Post, \{\}). & \text{Rule (8)} \\
scg\_i(Post, Prog, WPre, VCs) \leftarrow & \text{Rule (9)} \\
\quad decompose(Prog, Last, Remainder), & \\
\quad scg\_i(Post, Last, WPre_1, VCond_l) \wedge & \\
\quad scg\_i(WPre_1, Remainder, WPre, VCond\_r) \wedge & \\
\quad VCs = Cond_l \cup Cond\_r. & \\
scg\_i(Post, assign(LHS, RHS), WPre, VCs) \leftarrow & \text{Rule (6)} \\
\quad WPre = subs(Post, RHS = LHS) \wedge safe\_assign(LHS, RHS), & \\
scg\_i(Post, for(idx(I, Begin, End), Stmt, Invariant), WPre, VCs) \leftarrow & \\
\quad InvIterate = Inv[I+1/I], & \\
\quad scg\_i(InvIterate, Stmt, WPre, VC), & \\
\quad InvBegin = Inv[Begin/I], & \\
\quad VCs = \{((Inv \wedge (I \leq End)) \Rightarrow WPre), & \\
\qquad (Inv \wedge I \geq (End+1)) \Rightarrow Post\} \cup VC. &
\end{array}$$

Figure 6: Automatic Synthesis of Generic Safety Conditions

The execution of the meta-interpreter, given a program specification as input, is as follows:

- It works backwards through the program statement by statement applying the sequence rule for splitting a list of statements into the last statement and remainder list of statements and subsequently processes the single statement and the remainder list of statements.

- At each intermediary point, it tries to find a matching rule that generates the actual verification conditions, the weakest pre-condition, the safety expressions and subgoals.

- Finally, the condition that the actual pre-condition implies the weakest pre-condition is added to the set of verification conditions.

The process of safety condition generation is independent of the particular safety policy or logic used. Applying the meta-interpreter to the example program of Figure 3 generates the verification conditions and safety expressions as shown below.

5.2 Simplification

The process of synthesizing proof obligations is independent of the particular safety policy or logic used. The proof obligations have two separate conjuncts, one for functional correctness and another for the generic safety obligations. When verifying safety and correctness properties, fully automatic analysis is in general not feasible as it involves reasoning about arrays with mixed symbolic and numerical indices, type checking and arithmetics which in general is undecidable. Classical theorem provers are not well suited to symbolic arithmetic calculations which naturally arise from considerations about safety of program executions. Therefore, we are looking for simplification and reduction techniques that enable us to automate some proof steps and to solve the others interactively. The basic idea is to partition the safety and verification conditions into classes for which decision procedures and associated simple tool support exist and to analyze extensible sets of safety properties successively. If invariants, pre- and post-conditions are restricted to unquantified propositional formulas over algebraic expressions, combined by the usual connectives, then the simplification strategy could be based on the use of a computer algebra system. Computer algebra systems integrate features that are based on first-order equational logic to implement its evaluation mechanism. Standard facilities are substitutions and pattern matching in a wide variety of forms, automatic and user controlled simplification of arithmetic expressions and symbolic calculations involving operations on arrays and matrices. Moreover, they provide simple type checking procedures.

Substitution applied to the generated proof obligations and simplifying the expressions with the general data type axioms, cf. equations (1), (2), (3), (5), generates the following safety expression:

```
safe_assign(i_width, 300) and safe_assign(i_height, 350) and
exists(y, r_origin[1] + 1050 = y) and exists(x, r_origin[0] + 900 = x) and
safe_assign(d_area(0), im2, r_origin[0] + 300, r_origin[1] + 350) and
safe_assign(d_area(2), im0, r_origin[0] + 600, r_origin[1] + 700) and
safe_assign(d_area(1), im1, r_origin[0] + 900, r_origin[1] + 1050)
```

In other words, the program is correct with respect to its pre- post-conditions.

5.3 Domain-specific Safety Checking

Generally, in order to verify safety of program execution, taking into account its environment, it needs to be shown that all generic safety conditions follow from the actual safety specification Γ_{safe}.

$$\Gamma_{safe} \vdash \neg(\exists x : \neg SafeExpr(x) \wedge WPre \cup Conds)$$

Considering this particular example, it only needs to be shown that

$$\Gamma_{safe} \vdash \neg(\exists x : \neg SafeExpr(x)),$$

where Γ_{safe} is a domain-specific safety viewpoint as exemplified in Figure 5 and generally provided by the code consumer. A formal proof then essentially constitutes a formal statement that the program, when executed, will not violate any safety properties. Possible proof strategies are to use standard algorithms which are implemented in theorem provers or to submit specific proof obligations to specific tactics as the Omega solver. The Omega library is a set of routines for manipulating linear constraints over integer variables, Presburger formulas, and Integer tuple relations and sets [Pug92]. Constraint-logic programming combines the advantages of declarative logic programming with efficient constraint solving. It seems to be the most adequate choice to automate verification of the generated and simplified safety conditions. Implementing the domain-specific safety viewpoint Γ_{safe} specified in Figure 5 using a constraint-logic programming language and submitting the simplified safety expression automatically yields the result *no*. It corresponds with the obvious fact that the program execution is unsafe, taking into account the particular safety requirements modeled as finite domain-constraints.

6 Related Work

Numerous research systems allow the certification of compiled code of platform-dependent safety properties such as stack safety, accessed memory regions and control flow safety based upon appropriate type systems. The presented framework adopts ingredients from Necula's and Lee's approach (cf. [Nec97, NL98]) for proof-carrying code, but in this work safety properties can be analyzed on a higher code level than assembly language instructions through generic safety types. Touchstone is a compiler that translates programs written in a type-safe subset of the C programming language into assembly language programs, and a certifier that automatically checks the type safety and memory safety of the produced program. A verification condition generator produces a safety predicate for each function. The method applied combines forward symbolic evaluation and verification condition generation in contrast to the standard backward analysis for verification condition generation. The certifier performs further optimizations like array bounds-checking elimination.

Coq (cf. [FM99]) is a system for specifying and certifying imperative programs. The programs are given in an ML-Pascal like intermediate code mixing imperative features (references, arrays, while loops, sequences) and functional features. They are specified in a Floyd-Hoare logic style, by insertion of logical assertions, such as pre-conditions, post-conditions and loop invariants. Termination is justified by the insertion of a pair variant/relation associated to each loop or recursive function. Then an automatic tactic takes a specified program and produces some proof obligations, whose validity implies both correctness and termination of the initial program.

[CW00] presents a type system for specifying bounds on resource bound consumption and a method for certifying those bounds by considering cost functions. They provide a compiler generating certified executables from source code. [LPR01] uses abstract interpretation for proving consistency of higher-level domain-specific properties to a safety policy. Membership equational logic is used as logical framework for automated verification of frame safety properties.

A number of static analysis and certification approaches formulate verification problems as systems of symbolic and numerical constraints. [WFBA00] regards the detection of buffer overruns as an integer range analysis problem. [CKX01] combines a forward analysis to infer constraints at designated program points and a backward method for deriving a safety pre-condition. The derived pre-conditions are used for eliminating partially redundant checks and program optimization. Calculations are performed using the Omega library. [XMR00] presents a safety-checking analysis technique that only requires that the initial inputs to the untrusted program be annotated with typestate information and linear constraints. Polyspace is a commercial tool that allows the automatic detection of common programming errors for C code at compilation time. Based on abstract interpretation it detects common programming errors as for example “out of bounds array index”, “uninitialized variable”, “division by zero”, “overflow” or “underflow”.

7 Conclusions

This paper has provided a new three-stage framework towards the certification of simple imperative programs annotated with loop invariants, pre-, and post-conditions. An extension of the Floyd Hoare verification framework allows the automatic computation of generic safety conditions for annotated imperative programs and taking into account side effects. A corresponding meta-interpreter has been devised. Viewpoints assemble particular domain-specific safety requirements in terms of the generic safety predicates. The proof of the generic safety conditions, having provided a particular viewpoint as context, ensures safety of program execution at all intermediate states in a particular environment.

In a former version of AutoBayes, the safety policy was hard-coded in the way the annotations were generated within the synthesis schemas. The code was annotated to prove division-by-zero and array-bounds safety (cf.[FS02]) and then a verification condition generator was used for creating verification conditions which are resolved by a theorem prover. The advantage of separate and dynamic safety viewpoints is that the designer or the synthesis system need not add all the safety-related assertions to the program or synthesis schemes thus making it less cluttered and easier to make type-related changes to the safety policy. By this way, certificates are by comparison relatively compact, easy to produce and to verify. The architecture facilitiates synthesis of certified software from libraries of re-usable components and safety theories. Particular safety requirements can be represented exactly and explicitly as separate theories using and refining definitions entailed in standard libraries. Generic safety conditions for annotated programs need to be generated only once. When changes are made to the safety requirements, there is no need for a full re-verification of the program to prove that these changes don’t effect the safety of program execution. It only needs to be shown that the generic safety conditions are consistent with the actual safety theory.

Although this work explores in detail only rather simplified contextual safety constraints, it is believed that the approach developed so far provides a framework that will be applicable to the formal treatment of a much wider variety of semantic safety properties like general resource constraints. To enable these kinds of domain-specific safety checks, the safety viewpoint would augment the program declarations and entail more complex safety axiomatizations.

Acknowledgements This report is an outgrowth of work done jointly with Bernd Fischer and Johann Schumann within the AutoBayes project. Part of this work was done during the author's visit at NASA Ames Research Center. The views and conclusions contained in this paper are those of the author.

References

[BS02] Bernhard Beckert and Steffen Schlager. Integer arithmetic in the specification and verification of Java programs. In *Proceedings, Workshop on Tools for System Design and Verification (FM-TOOLS), Reisensburg, Germany*, pages 7–14, 2002.

[CKX01] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. Deriving pre-conditions for array bound check elimination. In *Second Symposium on Programs as Data Objects*, volume 2053, pages 2–24. Lecture Notes in Computer Science, 2001.

[Cou01] P. Cousot. Abstract interpretation based formal methods and future challenges. In R. Wilhelm, editor, *Informatics, 10 Years Back - 10 Years Ahead*, pages 138–156. Lecture Notes in Computer Science, 2001.

[CW00] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–198, 2000.

[Eus99] J. Eusterbrock. Composing reusable synthesis methods through graph-based viewpoints. In S. Hölldobler, editor, *Intellectics and Computational Logic, Papers in Honor of W. Bibel*, pages 143–158. Kluwer academic publishers b.v., 1999.

[Eus01] J. Eusterbrock. Knowledge mediation in the world-wide web based on labelled dags with attached constraints. *Electronic Transactions on Artificial Intelligence*, 5:http://www.ep.liu.se/ej/etai/2001/020/, Section D 2001.

[FM99] J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[Fra96] Ranan Fraer. Tracing the origins of verification conditions. In *Proceedings International Conference on Algebraic Methodology and Software Technology AMAST*, pages 241–255. Springer Verlag, Lecture Notes in Computer Science, 1996.

[FS02] Bernd Fischer and Johann Schumann. Combining program synthesis with automatic code certification(system description). In *Conference on Automated Deduction (CADE'02). Copenhagen*, page 2002. LNAI, Springer, 2002.

[FSW02] Bernd Fischer, Johann Schumann, and Mike Whalen. Synthesizing certified code. Technical Report 02.03, RIACS, 2002.

[HCEH96] R. Heckel, M. Conrad, G. Egger, and J. Hiemer. Automatic intergration of safety invariants into z specifications. In B. Buth, R. Berghammer, and J. Peleska, editors, *Proc. Workshop on Tools for System Development and Verification, Bremen, Germany*, volume 1, pages 70–83. BISS Monographs, Shaker Verlag, 1996.

[Hei96] Maritta Heisel. An approach to develop provably safe software. *High Integrity Systems*, 1(6):501–512, 1996.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–581, 1969.

[ILL75] S. Igarashi, R. L. London, and D. C. Luckham. Automatic Program Verification I: A Logical Basis and its Implementation. *Acta Informatica*, 4:145–182, 1975.

[LPR01] Michael Lowry, Thomas Pressburger, and Grigore Rosu. Certifying domain-specific policies. In M.S. Feather and M. Goedicke, editors, *Proceedings 16th International Conference Automated Software Engineering*, pages 118–125, 2001.

[LS79] D. Luckham and N. Suzuki. Verification of array, record, and pointer operations in pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, 1979.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, 1997.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In M. Ducasse, editor, *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation.* ACM SIGPLAN Notices, 1998.

[Nus97] B. Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, 1997.

[Pug92] W. Pugh. The omega test: A fast practical integer programming algorithm for dependence analysis. *Communication of the ACM*, 8:102–114, 1992.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[XMR00] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of the ACM SIGPLAN'00 conference on Programming language design and implementation May 2000*, volume 35, pages 70–82. ACM SIGPLAN Notices, 2000.