Innovations Syst Softw Eng (2006) 2:147–163
DOI 10.1007/s11334-006-0010-z

ORIGINAL PAPER

Symbolic approximation: an approach to verification in the large

Peter T. Breuer · Simon Pickin

Received: 2 July 2006 / Accepted: 14 August 2006 / Published online: 28 October 2006

Abstract This article describes *symbolic approximation*, a theoretical foundation for techniques evolved for large-scale verification – in particular, for post hoc verification of the C code in large-scale open-source projects such as the Linux kernel. The corresponding toolset's increasing maturity means that it is now capable of treating millions of lines of C code source in a few hours on very modest support platforms. In order to explicitly manage the state-space-explosion problem that bedevils model-checking approaches, we work with approximations to programs in a symbolic domain where approximation has a well-defined meaning. A more approximate program means being able to say less about what the program does, which means weaker logic for reasoning about the program. So we adjust the approximation by adjusting the applied logic. That guarantees a safe approximation (one which may generate false alarms but no false negatives) provided the logic used is weaker than the exact logic of C. We choose the logic to suit the analysis.

1 Introduction

In recent years, our group has developed a static analysis tool for use in the post hoc verification of properties in the Linux kernel ([3,4]) and other large open-source projects. Initially, it was a matter of some pride that the prototype could efficiently deal with some 30,000 lines or so of source code at a time, which was about the size of the source code of a small Linux kernel driver of some 500 lines or so of C code, once referenced header files had been included and all macros expanded.

But taking the development onwards to deal with first hundreds of thousands and then millions of lines of source code has not been merely a question of linear improvement – the tool had to be coupled with a logic compiler in order to allow the programming logic that it used to be reconfigured for different analyses, and the way the tool applied that logic to a parsed program syntax tree was made configurable via a trigger–action rule system, again compiled into the tool on demand. The coverage of the tool also had to be extended repeatedly to deal with unexpected C code constructions that the gcc C compiler allows and the Linux kernel makes use of. The analysis now copes with the mix of C and assembler in the Linux kernel source (and the analysis tool itself is written wholly in C, thus making it easy to compile and distribute in an open source environment; the code itself is licensed under an open source license).

We call the context in which we understand the techniques we have successfully employed for this setting *symbolic approximation*. The underlying idea is to place the semantics of a C (or other language) program into an abstract domain where an analysis using symbolic logic can be carried out. However, the abstract domain contains a native notion of the approximation of programs to one another, so we can decide how accurate – or inaccurate – to make the program being analysed with respect to the real program, and in what direction, in informed tradeoffs. These approximations are *abstract interpretations* [7] of the original, and one of them is an exact interpretation, although it is never used

P. T. Breuer (✉) · S. Pickin
Area de Ingenieria Telematica, Dpto. Ingenieria,
Universidad Carlos III de Madrid, Butarque 15,
Leganes, Madrid 28911, Spain
e-mail: ptb@inv.it.uc3m.es

S. Pickin
e-mail: spickin@it.uc3m.es

files checked:	1055	
alarms raised:	18	(5/1055 files)
false positives:	16/18	
real errors:	2/18	(2/1055 files)
time taken:	~24h	
LOC:	~700K	(unexpanded)

```
1 instances of sleep under spinlock
        in sound/isa/sb/sb16_csp.c
1 instances of sleep under spinlock
        in sound/oss/sequencer.c
6 instances of sleep under spinlock
        in net/bluetooth/rfcomm/tty.c
7 instances of sleep under spinlock
        in net/irda/irlmp.c
3 instances of sleep under spinlock
        in net/irda/irttp.c
```

Fig. 1 Testing for sleep under spinlock in the 2.6.3 Linux kernel

in practice, as it would be impossibly unwieldy and perhaps not expressible within the range of approximations that we can manage. The approximations of interest are those that we can express using our tools and which simplify the analysis.

After obtaining a representation of the C code in the abstract domain, the abstract states that result from running (in an abstract sense) the abstract program are subjected to simultaneous interpretations in several different perspectives, ultimately deriving good or bad (or other similar humanly meaningful qualifications) in various perspectives at each point in the code. The word perspective is chosen here to denote a projection that detects some condition such as reading memory before assigning to it, or calling a function that may sleep under lock, or accessing a file via a handle that has been closed, and so on. Each perspective looks at the same symbolic representation of the abstract state at each point in a different way, hence the name.

For example, Fig. 1 shows the result of checking about 1,000 (1,055) of the 6,294 C source files in the Linux 2.6.3 kernel for a condition known as "sleep under spinlock", which is a potential deadlock on a multiprocessor system (indeed, also on a uniprocessor system, but the same source code is usually compiled for a uniprocessor system in a different way which removes from the object code the locks that are dangerous). At that time, 2 years ago, the test run took 24 hours running on a 550 MHz (dual) symmetric multiprocessor (SMP) personal computer (PC) with 128 MB random-access memory (RAM). Now, 2 years later, the same run takes about 1/4 of the time (about 6 hours).

To explain the test a little, the property being looked for is a call to a function that can sleep (i.e., that can be scheduled out of the CPU) from a thread that holds a spinlock (a locking mechanism that causes a waiting thread to enter a busy loop until the lock is released to it) at the time of the call. Trying to take a locked spinlock on one CPU provokes a busy wait (known as a spin) that occupies the CPU until the spinlock is released on another CPU. If the thread that has locked the spinlock is scheduled out of its CPU while the lock is held, then the only thread that was intended to release the spinlock is not running. If by chance that thread is rescheduled into the CPU before any other thread tries to take the spinlock, then all is well, but if another thread tries for the spinlock first, then it will spin uselessly, occupying the CPU and keeping out the thread that would have released the spinlock. If yet another thread tries for the spinlock, then on a two-CPU SMP system, the machine is dead, with both CPUs spinning forever waiting for a lock that will never be released. Such vulnerabilities are denial-of-service opportunities that any user can exploit to take down a system. Two-CPU machines are also common – any Pentium 4 of 3.2 GHz or more has a hyper-threading core that presents itself as two CPUs.

Clearly, calling a function that may sleep while holding the lock on a spinlock is a serious matter. Yet the test detected three real cases of sleep under spinlock (out of 18 alarms raised) in the tested portion of the Linux 2.6.3 kernel source, and those abuses had remained undetected under the scrutiny of thousands of eyes for timescales of years (one of the faults was in the `snd_sb_csp_load()` function in `sb16_csp.c`, the Sound Blaster sound card driver; another was in the `midi_outc()` function in `sound/oss/sequencer.c`, in the sound sequence generator code; the latter problem would have made an SMP machine liable to deadlock when receiving mail for the administrator that triggered several "you have mail" system sound advisories at once – see Fig. 2). Two of the problems detected were monitored until they were detected and removed by contributors or maintainers in releases of the kernel about 6 months later, and the third (in `sequencer.c`) was not removed until the kernel maintainers were notified at release 2.6.12.5 of the kernel.

What problem are we solving by our approach? In the first instance, we are aiming to avoid the state-space-explosion problem that bedevils full model-checking techniques by deliberately targeting the notion of approximation, and understanding and controlling the way in which we approximate. People working with model checking and applying the techniques to similar repositories as we do (David Wagner and colleagues' work at Berkeley comes to mind, see for example [14], where Linux user space and kernel space memory

Fig. 2 Sleep under spinlock instances in kernel 2.6.3

File & function	Code fragment
sb/sb16_csp.c: snd_sb_csp_load	619 spin_lock_irqsave(&p->chip->reg_lock, flags); ... ... 632 unsigned char *kbuf, *_kbuf; 633 _kbuf = kbuf = kmalloc (size, GFP_KERNEL);
oss/sequencer.c: midi_outc	1219 spin_lock_irqsave(&lock,flags); 1220 while (n && !midi_devs[dev]->outputc(dev, data)) { 1221 interruptible_sleep_on_timeout(&seq_sleeper,HZ/25); 1222 n--; 1223 } 1224 spin_unlock_irqrestore(&lock,flags);

pointers are given different types, so that their use can be distinguished, and [15], where C strings are abstracted to a minimal and maximal length pair and operations on them abstracted to produce linear constraints on these numbers) also have to make approximations in order to apply their tools. Our approach is to make the approximation itself a rigorously understood area. The approach we have followed assigns a customisable abstract approximation semantics to C programs, via a customisable program logic for C. In general we decide to approximate on the safe side of reality, generating approximations in which we can only say that a program *may* do something bad, not that it *must* do something (good or bad). In that approach, any alarm sounded may err in possibly shouting where there is nothing to shout about, but will not be quiet when there is really something to be alarmed about (always within the parameters of the analysis itself – we cannot for example cope with self-altering C code, because the analysis inherently assumes the code does not change, and indeed that memory for particular data does not overstep memory areas for other data).

A more lightweight technique still is exemplified by Jeffrey Foster's work with CQual,[1] which extends the type system of C in a customisable manner. In particular, CQual has been used to detect double-spinlock takes, a subcase of one of the analyses performed by our tool.

The SLAM project [1] originating at Microsoft also analyses C programs using a mixture of model-checking, abstract interpretation and deduction in a way that is related to our approach (except that we do not use model-checking). That technology is an order or more of magnitude slower than ours, but also works by creating an abstraction of the program code, and also generates intermediate state descriptions mechnically. It differes in that it iterates at a global level, improving the fineness of its abstraction each time in order to successively rule out more false alarms, and we don not iterate.

[1] `http://www.cs.umd.edu/~jfoster/cqual/`. See [12,13].

Static analysis techniques generally abstract away some details of the program state, generating an abstract interpretation [7] of the program. Examples include ignoring a state in favour of the time taken to generate it, or restricting to a particular set of program variables (known as slicing), or looking only at the condition of certain logical assertions rather than the state itself. Abstract interpretation in a symbolic domain forms a fundamental part of the analysis here. In that domain, approximation in the sense of abstract interpretation can be defined exactly, and deliberate approximations on our part serve to simplify the description of the state that is propagated by the analysis; for example, "don't know" is a valid literal in the analysis domain, thus a program variable which may take any of the values 1, 2, or 3 in reality may be described as having the value "don't know" in the symbolic abstraction, leading to a state s described succinctly by one atomic proposition, not a disjunct of three. And although the logic of compound statements like `for`, `while`, etc. manipulates the logic of the component statements with genericity in the standard configuration, the logic of atomic statements like assignment is usually configured by the user to provide some simplification; for example an assignment to program variable `x` may be configured to delete references to the old value of `x` in the state, but not to assign a (particular) new value, thus giving an abstraction in which only the fact of assignment and reference is visible, not the value assigned or read.

Model checking [9] is the technique presently best know for semanitcally-based analyses. In model-checking a large transition graph is constructed to represent how the program may move from one state to another. The graph is then analysed to see if it satisfies required properties. However, even for simple programs or protocols, the data structures involved are so large that verifiction becomes intractable – the problem is know as "state-space explosion"

In practice, very large programs are either abstracted [10] or otherwise approximated or interpreted before model checking is applied. For example, a 32-bit integer variable may be represented as having just two

abstract states: zero, and non-zero. Our approach likewise abstracts the program and it takes it into a symbolic domain where approximation of programs also has a well-understood meaning.

The remainder of this article is structured as follows: the setting will be introduced slowly through Sects. 2, 3, 4 and 5; the full theory used will be described in Sect. 6, and the detail of the treatment of C will be given in Sect. 7, along with the definitions that customise the analysis. Section 8 will discuss how the analysis is interpreted through different perspectives; configurations of the analyser for a small variety of problems and the results are discussed in Sect. 9.

2 The simple approach

We introduce the concepts involved in our approach by way of an initial simple view of programs, based on classical Hoare semantics. In that approach a program fragment is represented by the relation the code establishes between a predicate $p \in P$ describing the initial state of the system before the code has run, and a predicate $q \in P$ describing the final state of the system after the code has run. That is, the representation falls in the space of relations with domain and co-domain P:

$$P \leftrightarrow P$$

where P is some sufficiently expressive domain of predicates on the finite state space of the C program, closed under logical conjunction and disjunction, and containing at least the atomic predicates involving equality, variables, and literal integer constants in the range of 32-bit integers, $[-2^{31}, 2^{31}-1]$. E.g. $(x = 1) \wedge (y = 2) \vee (z = 3)$ is a predicate.

Let T be a domain of terms representing at least subranges of integer elements from $[-2^{31}, 2^{31}-1]$. Terms (i.e. integer ranges) are compared via subset, the full range being the most underspecified term, written $\bot$, and the empty range being the overspecified term, written $\top$. Subset comparison $t_1 \supseteq t_2$ is written the other way round in terms of refinement, $t_1 \sqsubseteq t_2$ (i.e., t_2 refines t_1) , so that $\top$ (the empty range) is the top (most refined) element in the lattice ordered by refinement – "refinement is confinement" may be used to help remember which way round it goes. Terms include at least the single integer constants and variables representing single integer values, and have a formal finite meet operator (which may or may not be the set union – $[a,b] \sqcap [c,d] = [\min\{a,b\}, \max\{c,d\}]$ is an alternative, for example, when the terms include only the contiguous ranges).

Now let (P, T) be the space of paired predicates and terms, written $p \rhd t$. For example, $(x = 1) \wedge (y = 2) \rhd x$ represents a term defined over the range defined by $(x = 1) \wedge (y = 2)$, taking value x (i.e. 1) on that range; $(x = 1) \vee (z = 2) \rhd]0[$ represents a term defined over the range $(x = 1) \vee (z = 2)$ and it takes any value there from the whole range $[-2^{31}, 2^{31}-1]$ excluding 0.

There is then a ready-made notion of refinement on the domain (P, T) obtained by regarding $p \rhd t$ as being bound to t on p but bound to the empty range on the complement of p. That is, imagine that $P = \mathrm{P}(S)$, the power set for some state space S, and use the refinement induced by comparing the $p \rhd t$ as though they were the functions $s \mapsto t$ for $s \in p$, $s \mapsto \top$ for $s \notin p$, using the (pointwise) comparison from the function space T^S:

Definition 1 *The refinement ordering $\sqsubseteq$ on (P, T) is defined by*

$$p_1 \rhd t_2 \sqsupseteq p_2 \rhd t_2 \quad \textit{iff} \quad p_1 - p_2 \Rightarrow t_1 = \top \quad \textit{and } p_1 \wedge p_2 \Rightarrow t_1 \sqsupseteq t_2$$

□

This definition implies

$$\frac{p_1 \Rightarrow p_2}{p_1 \rhd t \sqsupseteq p_2 \rhd t} \qquad \frac{t_1 \sqsupseteq t_2}{p \rhd t_1 \sqsupseteq p \rhd t_2}$$

and thus this notion of refinement extends the refinement induced on the cross-product; i.e., if the predicate part is more confined, then the pair is more refined. If the term part is more confined, then the pair is more refined.

Lemma 1 *Refinement as defined above on (P, T) is a partial order relation, extending the product partial order relation induced from the refinements on P and T (refinement on P is implication, with* T *(true) being most unrefined and* F *("false") being most refined).*

We can now represent simple C programs by their Hoare semantics with respect to this domain, as relations in

$$(P, T) \leftrightarrow (P, T)$$

Example (Post-increment) x++ changes the value of x stored in the state, and returns the old value $x - 1$ of x:

$$\overline{p \rhd t \quad \mathtt{x{+}{+}}; \quad p[x-1/x] \rhd x-1}$$

In contrast, pre-increment ++x returns the new value of x:

$$\overline{p \rhd t \quad \mathtt{{+}{+}x}; \quad p[x-1/x] \rhd x}$$

In both cases, the initial returned value term t is discarded. □

Example The empty program has the action of the identity operator:

$$\frac{}{p \rhd t \quad ; \quad p \rhd t}$$

□

The result term is explicitly used in GNU C when a statement is made into an expression by surrounding the code with `({...})`, as for example in the macro that calculates the minimum of two values as follows in order to avoid double evaluation:

#define MIN(x, y)

$(\{\text{typeof}(x)_x = (x); \text{typeof}(y)_y = (y);\ _x <_y ?_x :_y;\})$

This macro is used in practice in order to guarantee that the generated code be inlined. By convention, programmers do not use variable names that start with an underscore, so the code is safe. The problem with writing instead the simpler `((x)<(y)?(x):(y))` is that the smaller of the expressions `x` and `y` will be evaluated twice, once in the comparison, and once as a result. Since either of `x` or `y` may be complicated expressions (e.g. `a+=2` for `x`) at the point where the macro is used, both the result and the side-effect may vary according to the number of evaluations of the macro parameters.

Some C code statements have always been able to be used as expressions with minor modification. For example, discarding the trailing semicolon from `x=x+1;` gives the valid C expression `x=x+1`, which may be used like any other expression and has the new value of `x` after the assignment (and which changes the program state to reflect the assignment). Thus, binding a term to represent the result of a statement, as well as that of an expression, is a natural thing to do.

Further, in GNU C, the result can be remembered from one statement to another. Consider the following GNU C (for gcc 2.95) expression-statement:

```
({ int x; x=1; goto foo; foo: })
```

Experiment will show that it returns the value 1, thus the value returned by the assignment early on in the block is remembered until the end of the block (note, however, that the gcc 2.95 semantics is ill-defined near here, since the very similar `(int x;x=1; goto foo; x=2; foo: )` returns 2. Labels at the end of compound statements have been outlawed in gcc 3.4 and 4.0, but adding *empty* statements instead perfectly satisfies both gcc 3.4 and 4.0, and still returns 1; try `(int x; x=1; ; ; )`, which illustrates how the empty statement preserves and remembers the bound result).

Note that over a finite state space at least, a relation R in $(P, T) \leftrightarrow (P, T)$ representing a Hoare semantics for a program is generated by a *strongest postcondition operator*, $\hat{R}$, a function:

$$p \; R \; q \quad \text{iff} \quad \hat{R}(p) \Rightarrow q$$

for some function $\hat{R}$ that returns a strongest postcondition q given the initial condition p. This is usually the convenient representation for practice. The semantic relation R has the property that it is closed with respect to loosening of its right-hand-side argument q and with respect to tightening of its left-hand-side argument p. That is:

$$\frac{p_2 \rhd t_2 \sqsupseteq p_1 \rhd t_1 \quad q_1 \rhd u_1 \sqsupseteq q_2 \rhd u_2 \quad p_1 \rhd t_1 \; R \; q_1 \rhd u_1}{p_2 \rhd t_2 \; R \; q_2 \rhd u_2} \tag{1}$$

or, in terms of the strongest postcondition operator:

$$p_1 \rhd t_1 \sqsubseteq p_2 \rhd t_2 \quad \Rightarrow \quad \hat{R}(p_1 \rhd t_1) \sqsubseteq \hat{R}(p_2 \rhd t_2) \tag{2}$$

It is also the case that semantics generated from a postcondition operator is at least well defined as a relation on every possible input condition:

$$\mathrm{F} \rhd \top \quad R \quad \mathrm{T} \rhd \bot \tag{3}$$

Via weakening, this means $\forall a. \exists b.\ a \; R \; b$. It will always be the case for real code that the stronger $\mathrm{F} \rhd \top \; R \; \mathrm{F} \rhd \top$ holds (dead code cannot rise up and run), but we do not insist on it as a fundamental axiom.

Often it is appropriate to describe the behaviour of code *piecewise*. In the case of the fragment `if (x>0) x++;else x--`, for example, the program variable `x` is incremented if it is initially positive, and decremented if initially zero or negative. The appropriate symbolic representation is

$$\frac{}{p \rhd t \;\; \texttt{if (x>0) x++; else x--;} \;\; \begin{cases} x > 1 \wedge p[x-1/x] \rhd x-1 \\ x < 0 \wedge p[x+1/x] \rhd x+1 \end{cases}}$$

where the right-hand side expresses the semantics piecewise.

To formally express piecewise behaviours, we use the symbol $[\!]$ (disjunction) to denote the composition piecewise by parts, and form a commutative partially ordered algebra based on the domain (P, T):

$$A(P, T, [\!], \sqsubseteq)$$

[for short, $A(P, T)$] whose terms consist of the pairs from (P, T) combined using the disjunction symbol, with partial order as described in the paragraphs below. The following domain then forms the appropriate basic setting for the symbolic semantics of C programs:

$$(P, T) \leftrightarrow A(P, T)$$

Example In this domain, the strongest postcondition semantics of the fragment `if(x>0) x++; else x--`, is written as follows:

$$p \rhd t \quad \texttt{if (x>0) x++; else x--;} \quad \begin{array}{l} x > 1 \wedge p[x-1/x] \rhd x-1 \\ x < 0 \wedge p[x+1/x] \rhd x+1 \end{array}$$

□

Example As an example of using disjunction in a non-trivial way in a semantics specification, suppose that C function `trylock` either increments the global variable `x` and returns 1, or leaves it unchanged and returns 0. Its semantics are

$$p \rhd t \quad \texttt{trylock(\&x);} \quad p[x-1/x] \rhd 1 \,[\!]\, p \rhd 0$$

This represents the idea that we do not know what `trylock` (lock attempt) will do, whether it will succeed in obtaining a lock or not, until it is tried. Internally, the function may perhaps test the value of `x` or perform other operations that we could unravel, but the specification given approximates it as a black box via the idea that it either gets the lock and returns 1 to the outside world, or does not and returns 0. □

We extend the refinement relation through finite disjunctions by considering that $p_1 \rhd t_1 \,[\!]\, p_2 \rhd t_2$ has bound value $\top$ (void) on the complement of $p_1 \vee p_2$, bound value t_1 on $p_1 - p_2$, t_2 on $p_2 - p_1$, and the joint value $t_1 \sqcap t_2$ (that is, $t_1 \cup t_2$ when meet in the refinement ordering is set-theoretic union) on $p_1 \wedge p_2$. Then:

Definition 2 *The extension of the refinement relation to $A(P,T)$ is given by:*

$$p \rhd t \sqsupseteq [\!]_i\, p_i \rhd t_i \qquad \textit{iff} \qquad \begin{cases} p - \vee_i p_i \Rightarrow t \sqsupseteq \top \\ p \wedge (\wedge_j p_j - \vee_k p_k) \Rightarrow t \sqsupseteq \sqcap_j t_j \end{cases}$$

where the j, k separate the (finite) set of the i into two disjoint subsets, and

$$a \sqsubseteq [\!]_i\, p_i \rhd t_i \qquad \textit{iff} \qquad \forall i.\ a \sqsubseteq p_i \rhd t_i$$

To do the generic comparison $a \sqsubseteq b$ one decomposes b into its disjunctive components $p_i \rhd t_i$ and compares $a \sqsubseteq p_i \rhd t_i$ for each of them. □

Recall that $t_1 \sqsupseteq t_2$ means $t_1 \subseteq t_2$ (more confinement means more refinement). The first half of the definition is natural in practice; for example, in the case of three components, the disjunct takes value $\top$ on the complement of $p_1 \vee p_2 \vee p_3$, value t_1 on $p_1 - (p_2 \vee p_3)$, t_2 on $p_2 - (p_1 \vee p_3)$, t_3 on $p_3 - (p_1 \vee p_2)$, value $t_1 \sqcap t_2$ on $p_1 \wedge p_2 - p_3$, $t_2 \sqcap t_3$ on $p_2 \wedge p_3 - p_1$, $t_1 \sqcap t_3$ on $p_1 \wedge p_3 - p_2$, and value $t_1 \sqcap t_2 \sqcap t_3$ on $p_1 \wedge p_2 \wedge p_3$, inducing the appropriate comparisons.

Lemma 2 (i) *Refinement as defined above on the domain $A(P,T)$ is a partial order relation and extends the order defined on (P,T)*; (ii) *disjunction as defined on that domain is a commutative associative operation well defined with respect to the equality induced by refinement (and its zero is $\mathrm{F} \rhd \top$ – which equals $p \rhd \top$ for any p)*; (iii) *disjunction gives the greatest lower bound of the disjuncts with respect to refinement in $A(P,T)$.*

As remarked, we expect all Hoare-style semantic relations F in the extended domain with disjunction to respect (1) in the new setting. The expectation is reasonable, thinking of pairs $p \rhd t$ as (constant-t) partial functions over state with domain p, because further refinement specifies further the bound term, or the possible states, resulting in a more specified result of an applied computation.

It is now possible to represent exactly any operation on a finite state space.

Example To represent multiplication, `x*z`, one merely has to enumerate all the possible values separately, as a large disjunct:

$$p \quad \texttt{x*y} \quad p \wedge x = 1 \wedge y = 1 \rhd 1 \,[\!]\, p \wedge x = 2 \wedge y = 1 \rhd 2 \,[\!]\, \cdots$$

but this level of detail in the description is not feasible in practice. In practice we would perhaps provide a symbolic approximation of the form

$$p \quad \texttt{x*y} \quad p \wedge 0 \le x < 10 \wedge 0 \le y < 10 \rhd [0,99] \,[\!]\, \cdots$$

thus dividing the range up into decades, and stating in which decade the result must fall given the decades of the arguments. Or we might simply choose to record the sign:

$$p \quad \texttt{x*y} \quad p \wedge 0 \le x \wedge 0 \le y \vee 0 \ge x \wedge 0 \ge y \rhd [0,+\infty] \,[\!]\, \cdots$$

Any way of representing multiplication that approximates (in the sense of $\sqsubseteq$) the exactly intended result is admissible. □

To deal conveniently and formally with programs that give rise to a disjunction in the symbolic representation of their semantics, we introduce logical rules that allow the disjunctive components to be considered separately:

$$\frac{p \rhd t \quad \mathrm{a} \quad f \qquad p \rhd t \quad \mathrm{a} \quad g}{p \rhd t \quad \mathrm{a} \quad f \,[\!]\, g} \tag{4}$$

and conversely:

$$\frac{p \rhd t \quad \mathrm{a} \quad f \,[\!]\, g}{p \rhd t \quad \mathrm{a} \quad f} \qquad \frac{p \rhd t \quad \mathrm{a} \quad f \,[\!]\, g}{p \rhd t \quad \mathrm{a} \quad g} \tag{5}$$

These rules merely state that semantic relations R arising from programs a are *filters* on their right-hand side. That is, as well as being closed under weakening on the right, the relation is closed under finite meet [i.e. disjunction, since that gives the greatest lower bound in $A(P,T)$] on the right. This makes sense in terms of a pure Hoare semantics, since if program a necessarily reaches termination condition q_1 starting from initial condition p and it also necessarily reaches termination condition q_2 starting from p, then it reaches termination condition $q_1 \wedge q_2$. In the present setting, however, $p \rhd t$ a f should be understood as the claim that it *is possible* that end-condition f may arise, in some trace that program a will give rise to, or EF f in the notation of temporal logic (CTL). Then $f \,[\!]\, g$ is EF $f \wedge$ EF g, i.e. it is both possible that f may arise *and* it is possible that g may arise, so it makes sense that disjunction on the right should behave like a conjunction.

3 Program composition in the simple approach

Table 1 gives the simple Hoare-style semantics of the principal C program constructors as relations in $(P,T) \leftrightarrow A(P,T)$. The sequence constructor is composition of relations, and the empty statement is the identity embedding. Usually sequential composition entails dropping the result returned by the first statement for that returned by the second statement, given the semantics of the individual components. But that is not the case when one of the statements is the empty statement, which always borrows the remembered result as its returned value.

`If` statements produce disjunctions, bound to the term literal $\top$ signifying an invalid result – in GNU C the type of the value delivered by an `if` statement or `while` loop is `void`, and it cannot be returned as a result in an expression:

```
int y = ({ int x; x=1; while(0); });
 test.c:...: void value not
 ignored as it ought to be
```

For the test x in the `if`, let x be a new condition variable which represents the value returned by x. Rule (8) means to let q_1 be conditions that are true when the test returns a 1 (or any other nonzero value) and q_0 be conditions that are true when the test returns 0 (in an ideal world, the strongest such), so-called *branch hypotheses*. They appear also in rule (9) for a `while` loop.

How is an invariant condition p' for the loop discovered in practice? First of all note that there is such a p', since T (true) will do (we can validly set both q_1 and q_0 to T too, in the eventuality of our complete ignorance). So the question is how to get hold of a good invariant.

Starting from p, we first calculate suitable branch hypotheses q_0 and q_1, as set out in the next section, and then try the p' calculated from

$$q_1 \rhd x \quad \mathtt{a;} \quad p' \rhd t_1$$

If this $p' \Rightarrow p$, then p itself is an invariant. Otherwise we replace p with $p \vee p'$ and try again. If this is an invariant, then we are done. If not, we write $p \vee p'$ in disjunctive normal form and erase components of the conjuncts in the disjuncts one by one, testing each time to see if we have an invariant. The procedure ends after a finite number of steps. At the very worst it terminates with T, which is an invariant (though normally not a useful one).

Note that the existential quantification which appears in rule (10) can be replaced by a (large) finite disjunction on a finite state space. In practice, there are likely either no appearances of the bound variable in the quantified predicate, so the point is moot, or the assignment is an increment or other simple change that can be effected by substituting one term for another in the predicate. For example, if the assignment is `x=x+1;`, then the predicate on the right of the conclusion in the rule can be $q[x-1/x]$.

4 Generating good branch hypotheses

In practice, the predicates q_0, q_1 (call them *negative* and *positive*, respectively) that say what hypotheses can be brought to bear when going down the `else` or `then` branches of an `if`, respectively (and for breaking out of, or staying in, respectively, a `while` loop) are generated from the syntax of the test expression x. For example, if x is just the program variable `x`, then q_1 and q_0 are trivial (i.e. essentially just the same p as the input condition), because

$$\frac{}{p \rhd t \quad \mathtt{x} \quad (x \neq 0) \wedge p \rhd]0[\;[\!]\; (x = 0) \wedge p \rhd 0}$$

In general, we will set $q_1 = \mathbf{pos}(p, \mathrm{e})$, for expression e, and $q_0 = \mathbf{neg}(p, \mathrm{e})$, and aim for

$$\frac{}{p \rhd t \quad \mathrm{e} \quad \mathbf{pos}(p,\mathrm{e}) \rhd]0[\;[\!]\; \mathbf{neg}(p,\mathrm{e}) \rhd 0} \tag{11}$$

Table 2 shows the major expression constructions and their decompositions.

Many other propositional expressions in C code may also translate easily, depending on precisely what is in the class of predicates P (we may assume that it contains

Table 1 The simple Hoare-style logic of C constructors

$$\frac{p \rhd t_1 \ \mathtt{a} \ q \rhd t_2 \qquad q \rhd t_2 \ \mathtt{b} \ r \rhd t_3}{p \rhd t_1 \ \mathtt{a;b;} \ r \rhd t_3} \tag{6}$$

$$\frac{}{p \rhd t \quad ; \quad p \rhd t} \tag{7}$$

$$\frac{p \rhd t \quad \mathtt{x} \quad (q_1 \rhd]0[\ [\!] \ q_0 \rhd 0) \qquad q_1 \rhd x \quad \mathtt{a;} \quad r_1 \rhd t_1 \qquad q_0 \rhd x \quad \mathtt{b;} \quad r_0 \rhd t_0}{p \rhd t \quad \mathtt{if\ (x)\ a;\ else\ b;} \quad r_1 \vee r_0 \rhd \top} \tag{8}$$

$$\frac{p' \rhd t \quad \mathtt{x} \quad (q_1 \rhd]0[\ [\!] \ q_0 \rhd 0) \qquad q_1 \rhd x \quad \mathtt{a;} \quad p' \rhd t_1 \qquad p \Rightarrow p'}{p \rhd t \quad \mathtt{while\ (x)\ a;} \quad q_0 \rhd \top} \tag{9}$$

$$\frac{p \rhd t \quad \mathtt{e} \quad q \rhd s}{p \rhd t \quad \mathtt{x = e;} \quad \exists \zeta . q[\zeta/x] \wedge x = s[\zeta/x] \rhd x} \tag{10}$$

Table 2 Generating branch hypotheses from expressions.

$$\begin{array}{ll} \mathbf{pos}(p,\mathrm{x}) = (x \neq 0) \wedge p & \mathbf{pos}(p,\mathrm{x{>}k}) = (x > k) \wedge p \\ \mathbf{neg}(p,\mathrm{x}) = (x = 0) \wedge p & \mathbf{neg}(p,\mathrm{x{>}k}) = (x \leq k) \wedge p \\ \\ \mathbf{pos}(p,\mathtt{!e}) = \mathbf{neg}(p,\mathrm{e}) & \mathbf{pos}(p,\mathrm{x{+}{+}}) = (x \neq 1) \wedge p[x-1/x] \\ \mathbf{neg}(p,\mathtt{!e}) = \mathbf{pos}(p,\mathrm{e}) & \mathbf{neg}(p,\mathrm{x{+}{+}}) = (x = 1) \wedge p[x-1/x] \\ \mathbf{pos}(p,\mathrm{a\&\&b}) = \mathbf{pos}(\mathbf{pos}(p,\mathrm{a}),\mathrm{b}) & \mathbf{pos}(p,\mathrm{a||b}) = \mathbf{pos}(p,\mathrm{a}) \vee \mathbf{pos}(\mathbf{neg}(p,\mathrm{a}),\mathrm{b}) \\ \mathbf{neg}(p,\mathrm{a\&\&b}) = \mathbf{neg}(p,\mathrm{a}) \vee \mathbf{neg}(\mathbf{pos}(p,\mathrm{a}),\mathrm{b}) & \mathbf{neg}(p,\mathrm{a||b}) = \mathbf{neg}(\mathbf{neg}(p,\mathrm{a}),\mathrm{b}) \\ \multicolumn{2}{c}{\mathbf{pos}(p,\mathrm{e?a:b}) = \mathbf{pos}(\mathbf{pos}(p,\mathrm{e}),\mathrm{a}) \vee \mathbf{pos}(\mathbf{neg}(p,\mathrm{e}),\mathrm{b})} \\ \multicolumn{2}{c}{\mathbf{neg}(p,\mathrm{e?a:b}) = \mathbf{neg}(\mathbf{pos}(p,\mathrm{e}),\mathrm{a}) \vee \mathbf{neg}(\mathbf{neg}(p,\mathrm{e}),\mathrm{b})} \end{array}$$

at least the atomic order and equality predicates comparing variables and constants). In our implementation, we are able to handle exactly only comparisons between variables and constants.

However, one may choose to make no use whatever of the information from the test in the branches of the `if` statement and use the following default:

$$\frac{}{p \rhd t \quad \mathtt{e} \quad \mathrm{T} \rhd]0[\ [\!] \ \mathrm{T} \rhd 0} \tag{12}$$

i.e. $\mathbf{pos}(p,e) = \mathbf{neg}(p,e) = \mathrm{T}$ (true). This rule stops any recursive search.

Example When the `if` statement has the form `if(x++<=0) a;else b;`, the complete semantic rule for it is:

$$\frac{\begin{array}{ll}(x-1 \leq 0) \wedge p[x-1/x] \rhd 1 & \mathtt{a;} \quad r_1 \rhd t_1 \\ (x-1 > 0) \wedge p[x-1/x] \rhd 0 & \mathtt{b;} \quad r_0 \rhd t_0\end{array}}{p \rhd t \quad \mathtt{if\ (x++<=0)\ a;\ else\ b;} \quad r_1 \cup r_0 \rhd \top}$$

□

The definitions above lend themselves to an inductive proof that when the expression e comes out nonzero in real life, under initial conditions p, then $\mathbf{pos}(p,\mathrm{e})$ is true, and similarly for **neg**.

Lemma 3 *The definitions of* **pos** *and* **neg** *given reflect the real semantics of the C expression constructors described, in that whenever expression e comes out nonzero under conditions p, then* $\mathbf{pos}(p,e)$ *is true, and whenever e comes out zero under conditions p, then* $\mathbf{neg}(p,e)$ *is true.*

Proof This is true for the induction-stopper $\mathbf{pos}(p,e) = \mathbf{neg}(p,e) = \mathrm{T}$. If it is true for e, then it is true, for example, for `!e` because whenever `!e` is nonzero, it is because e is zero and by induction $\mathbf{neg}(p,\mathrm{e})$ holds, which is $\mathbf{pos}(p,\mathtt{!e})$ by definition. Similarly, whenever `!e` is zero, it is because e is nonzero and by induction $\mathbf{pos}(p,\mathrm{e})$ holds, which is $\mathbf{neg}(p,\mathtt{!e})$ by definition.

In the case of a&&b, for example, when this is nonzero it is because a is nonzero, meaning that, inductively, $q = \mathbf{pos}(p,\mathrm{a})$ initially holds, and because b is also subsequently nonzero, meaning that $\mathbf{pos}(q,\mathrm{b})$ subsequently holds, which is $\mathbf{pos}(p,\mathrm{a\&\&b})$ by definition. All the other cases are similar. □

5 What is meant by symbolic approximation?

The statement that a given semantics $\lceil - \rceil_1$ [a map from C programs to the relations in $(P,T) \leftrightarrow A(P,T)$ that respect (1)] is an approximation in this setting can be given a solid formal basis. The formal assertion is that, for all programs a,

$$\lceil \mathrm{a} \rceil_1 \subseteq \lceil \mathrm{a} \rceil_2 \tag{13}$$

where the left-hand side of the inequality is the semantics of C code a as set out in the purported approximation, and the right-hand side is the exact semantics of the C code in the same setting (which in principle, but not as a practical matter, can be expressed point by point over the finite state in the program).

Since the semantic relation $\lceil \mathrm{a} \rceil_1$ is populated by a deduction system, what is claimed by (13) is *soundness*: the deduction system in question cannot establish more truths about any program a than are present in exact reality.

Extending the inequation (13) to the situation where $\lceil - \rceil_2$ is not an exact semantics of C, but instead some other semantics given by some other logical deduction system, the inequation asserts *relative soundness*, i.e. that the logical deduction system associated with $\lceil - \rceil_1$ cannot produce more truths about an arbitrary program a than are produced by the logical deduction system associated with $\lceil - \rceil_2$.

We take this as the definition of what it means for one semantics to be more approximate than another.

Definition 3 *Semantics $\lceil - \rceil_i$, $i = 1, 2$ mapping programs a to $\lceil a \rceil_i$ in the domain $(P, T) \leftrightarrow A(P, T)$ are said to satisfy the approximation relation*

$$\lceil - \rceil_1 \sqsubseteq \lceil - \rceil_2$$

if $\lceil a \rceil_1 \subseteq \lceil a \rceil_2$ as relations for each a.

We call this the *relative soundness* of the first interpretation with respect to the second. Relative soundness (with respect to the exact semantics) means the exact semantics can prove more statements about the program behaviour. The idea is that the approximating semantics knows less about what the program will do, and correspondingly, can say less about it.

The claim (13) that a given semantics approximates the real semantics in particular, can be proved for a *compositional* semantics by induction on the construction of the syntax of a.

Definition 4 *A semantic interpretation $\lceil a \rceil$ in $(P, T) \leftrightarrow A(P, T)$ is compositional, if for every syntactic constructor F, there is a semantic transformer $\lceil F \rceil$ with*

$$\lceil F(b) \rceil = \lceil F \rceil (\lceil b \rceil) \tag{14}$$

Note that the constructor must preserve the property of respecting (1).

When we know more about part of a program, then we know more about the whole program:

Definition 5 *A semantic constructor F is* monotonic *if, for arbitrary semantics x, y in $(P, T) \leftrightarrow A(P, T)$, $x \sqsubseteq y$ implies $\lceil F \rceil (x) \sqsubseteq \lceil F \rceil (y)$.*

Lemma 4 *It is sufficient for compositional semantics $\lceil a \rceil_1$, $\lceil a \rceil_2$ in order to show relative soundness between them that, for each syntactic constructor F:*

(i) *$\lceil F \rceil_1 (x) \sqsubseteq \lceil F \rceil_2 (x)$ for arbitrary semantics x, and*
(ii) *either or both of $\lceil F \rceil_1$, $\lceil F \rceil_2$ are monotonic semantic constructors.*

Proof Suppose $x \sqsubseteq y$ for semantic relations x, y. With (i) one gets both (a) $\lceil F \rceil_1 (x) \sqsubseteq \lceil F \rceil_2 (x)$ and (b) $\lceil F \rceil_1 (y) \sqsubseteq \lceil F \rceil_2 (y)$, and if (ii) gives one $\lceil F \rceil_2 (x) \sqsubseteq \lceil F \rceil_2 (y)$, one can use (a), while if (ii) gives one $\lceil F \rceil_1 (x) \sqsubseteq \lceil F \rceil_1 (y)$, one can use (b), thus obtaining

$$x \sqsubseteq y \Rightarrow \lceil F \rceil_1 (x) \sqsubseteq \lceil F \rceil_2 (y)$$

For induction, suppose that $\lceil \mathrm{a} \rceil_1 \sqsubseteq \lceil \mathrm{a} \rceil_2$. The above then gives

$$\lceil F \rceil_1 (\lceil \mathrm{a} \rceil_1) \sqsubseteq \lceil F \rceil_2 (\lceil \mathrm{a} \rceil_2)$$

which by compositionality is

$$\lceil F(\mathrm{a}) \rceil_1 \sqsubseteq \lceil F(\mathrm{a}) \rceil_2$$

which carries $\lceil \mathrm{a} \rceil_1 \sqsubseteq \lceil \mathrm{a} \rceil_2$ through every construction F of the language. □

Lemma 5 *Any compositional semantic interpretation $\lceil a \rceil$ is monotonic* (*with respect to refinement*).

Proof This is simply the case because, by definition, each semantic constructor $\lceil F \rceil$ creates a semantic relation [i.e. one satisfying (1)] when its argument does, which means monotonicity with respect to the refinement relation on pairs $p \rhd t$. □

Recall that the interpretations of interest here are defined by what can be deduced about programs in a given logical deduction system. That is

$$\lceil \mathrm{a} \rceil = \{(p \rhd t, a) : \ \vdash p \rhd t \ \mathrm{a} \ a\}$$

It is the case here that $p \rhd t \ \mathrm{a} \ a$, and hence membership of $\lceil \mathrm{a} \rceil$, is relatively decidable, in the sense of being reducible to a question of pure logic not involving programs, but this will not be proved. Then:

Lemma 6 *Any interpretation given by a system of logical rules for deducing when $p \rhd u \ a \ q \rhd v$ which admits (1) provides a semantic relation.*

Proof The rule (1) explicitly provides for strengthening on the left and weakening on the right in each derived relation pair. □

Lemma 7 *In interpretations created by systems of logical deduction rules for $p \rhd u \quad a \quad q \rhd v$, the weaker logical system gives a more approximate interpretation.*

Proof The weaker system generates emptier relations $p \rhd u$ a $q \rhd v$, which is what "more approximate" means. □

Example Consider a rough semantics for an `if(t<0)` statement expressed in terms of its `then` and `else` branches a and b, with semantics x and y, respectively:

$$\lceil \texttt{if(t} < \texttt{0)} \rceil_1(x,y) = \{(p \rhd u, q \rhd \top) : (p \rhd 1)\ x\ (q \rhd s_1) \\ \wedge (p \rhd 0)\ y\ (q \rhd s_0)\}$$

The exact semantics has $p = p_0 \cup p_1$ disjointly, where p_0 makes the test `t<0` come out false, and p_1 makes it come out true, then requires that $q \rhd s_0$ be deducible from $p_0 \rhd 0$ down the `else` branch and $q \rhd s_1$ be deducible from $p_1 \rhd 1$ down the `then` branch:

$$\lceil \texttt{if(t} < \texttt{0)} \rceil_2(x,y) = \{(p \rhd u, q \rhd \top) : (p_1 \rhd 1)\ x\ (q \rhd s_1) \\ \wedge (p_0 \rhd 0)\ y\ (q \rhd s_0)\}$$

where $p_1 = p \wedge t < 0$ and $p_0 = p \wedge t \geq 0$.

The second interpretation (the exact semantics) gives the bigger set. If one takes $(p \rhd u, q \rhd \top)$ from the first set, then $(p \rhd 1)\ x\ (q \rhd s_1)$ and $(p \rhd 0)\ y\ (q \rhd s_0)$.

Now

$$p_0 \Rightarrow p \quad \text{and} \quad p_1 \Rightarrow p$$

and so

$$p_0 \rhd 0 \sqsupseteq p \rhd 0 \quad \text{and} \quad p_1 \rhd 1 \sqsupseteq p \rhd 1$$

and we suppose x and y, being semantics, respect (1), so

$$(p \rhd 0) y (q \rhd s_0) \Rightarrow (p_0 \rhd 0) y (q \rhd s_0) \quad \text{and} \\ (p \rhd 1) x (q \rhd s_1) \Rightarrow (p_1 \rhd 1) x (q \rhd s_1)$$

which means that $(p_1 \rhd 1)\ x\ (q \rhd s_1)$ and $(p_0 \rhd 0)\ x\ (q \rhd s_0)$, which puts $(p \rhd u, q \rhd \top)$ in the second set, and establishes that the first interpretation is an approximation to the second (the exact semantics). □

As to whether p_0 and p_1 in the example calculation above are always expressible within the set of predicates allowed, for any test condition, at worst one may enumerate one by one the set of points in the (finite) state where the test respectively comes out false and true, and construct the predicates that way.

There is thus nothing special about the test `t<0` in the above reasoning, except that the absence of consideration of a side-effect made for a shorter proof text. If the test in the `if` statement has a side-effect, one lets p_0 describe the state after the side-effect has occurred subject to the condition that the test comes out false, and lets p_1 also describe the state after the side-effect, but subject to the condition that the test comes out true.

It is an exercise to prove

Proposition 1 *The semantics* (*as defined in* (6), (7), (8), (9), (10), …) *of C code is compositional with semantic constructors that produce semantics that respect* (1) *and are monotonic in the semantics they take as arguments.*

Proof The table gives the constructors, and the logical deduction rules connected with each, each of which respect (1). The question of monotonicity is ostensibly nontrivial in the case of conditionals and loops, however, because the branch hypotheses p_0, p_1, and loop invariant p' (respectively) are generated by computational algorithms not explicitly given in the table. The branch hypotheses for a conditional, however, are generated independently of the syntax or semantics of the branches (using the test condition syntax alone as a guide) and the branch semantics are subsequently applied to these conditions to generate new predicates r_0, r_1. If the branch semantics are weakened (in the sense of the approximation), then they can only generate fewer of the same or weaker new predicates (since the set of predicates produced is closed under weakening), which leads to generating fewer of the possible results $r_1 \vee r_0 \rhd \top$ for the conditional construction semantics.

The loop invariant, however, is constructed by trying out the loop body semantics on the branch hypothesis q_1, so it might in principle depend non-monotonically in some way on that semantics. However, weakening the loop body semantics just produces fewer (and weaker) conditions p' to try as an invariant. So the resulting candidate p' is inevitably weaker than it might otherwise have been. Our process for finding an invariant given a candidate (as described in the text) is monotonic in the candidate, so it terminates with at worst a weaker invariant, which (since the set of predicates produced is closed under weakening) means the result generated by the loop construction semantics is just one of the possibilities created given the stronger loop body semantics, which is the required approximation result. □

Proposition 2 *The approximation semantics set out via Table* 1 *is an* (*over-estimating*) *approximation to the exact semantics of the corresponding subset of C.*

Proof It suffices to note that the logic in Table 1 is weaker than that of the exact semantics of C, because of the generation of positive and negative conditions arising from the test e in the `if` that are generally cruder than the exact p_0, p_1 that form the branch hypotheses in real life, and for the use of weaker loop invariants than in real life. Then apply Lemma 6. □

That leaves statements which change the flow of control, `break`, `return`, `goto` etc., which will be dealt with in the next section. The present section has been dedicated to a simpler universe, trying to show there that

there is a meaning to the notion of symbolic approximation. It turns out the notion of approximation in the symbolic domain amounts to giving weaker (or stronger) logical deduction rules setting out the semantics of programs as predicate (and bound term) transformers (or relations, in the wider realm).

6 Black box, grey box

Thus far, programs have been thought of as classical black boxes, which take an input, and produce an output; in terms of their effect on state, they accept an initial program state, run, and leave a finalised state on termination. Now they will be opened up for (limited) external observation of their internals as they run. A *grey box* is a program description in which one can metaphorically hit the stop button at some defined points during execution and observe the intermediate program state.

For C programs, the points at which we can observe the state during execution are strictly defined via certain *exceptional program exits*. The *normal* (not exceptional) program exit occurs when execution comes to the end of a program fragment, and there are three types of exceptional exits: a *return* exit, caused by a program hitting a `return` instruction in a subroutine; a *break* exit, caused by program execution hitting a `break` instruction inside a `while`, `for` or `do` loop, and a *goto* exit, caused by hitting a `goto` instruction that causes a jump out of the program.

These considerations give rise to a logic normal, return, break, goto (NRBG) that extends the normal logic set out in the previous section. See Fig. 3 for a graphical representation of the flows captured in the sequential and loop parts of the logic. We now write the logical rules from Sect. 2 with an extra N: qualifier

$$p \rhd t \quad \text{N: a} \quad q \rhd s$$

so that, for example, the rule for sequence reads:

$$\frac{p \rhd t_1 \ \text{N: a}\ q \rhd t_2 \quad q \rhd t_2 \text{N: b}\ r \rhd t_3}{p \rhd t_1 \ \text{N: a;b;}\ r \rhd t_3}$$

The N ("normal") part of the logic shown in this rule represents the way code flows by falling off the end of one fragment and into another, in sequence. To exit normally with r, the program must flow normally through fragment a, hitting an intermediate condition q, and then enter fragment b and exit it normally with r.

In contrast, the R part of the sequence logic (Table 3, rule (15)) represents the way code flows out of the sequential parts of a program through a return path. Thus, if q is the intermediate condition that is attained after normal termination of a, then one may either return from program fragment a with r, or else terminate a normally with q, then enter fragment b and return from b with r.

The logic of break is (in the case of sequence) exactly equal to that of return (Table 3, rule (16)). Where break and return logic do differ is in the treatment of loops. First of all, one may return from a *forever* `while` loop by returning from its body:

$$\frac{p \rhd 1 \quad \text{R: a;} \quad q \rhd \bot}{p \rhd t \quad \text{R: while(1) a;} \quad q \rhd \top}$$

Or one may go round the loop once, and then return:

$$\frac{p \rhd 1 \quad \text{N: a;} \quad q \rhd \bot \qquad q \rhd 1 \quad \text{R: a;} \quad r \rhd \bot}{p \rhd t \quad \text{R: while(1) a;} \quad r \rhd \top}$$

The general rule is given in (17) where p' is a loop invariant implied by p. We discussed in the last section how to derive p'.

On the other hand, (counterintuitively at first reading) there is no way of leaving a forever `while` loop via a break exit, because a break in the body of the loop causes a normal exit from the loop itself, not a break exit. The normal exit from a forever loop is by break from its body, as rule (18) says.

In a typical application, extra rules deal with code of special interest to the analysis. For example, if the precondition p is the claim that a spinlock count (the total number of locks taken and held so far) ρ is below or equal to n, for some n: $\rho \leq n$. Passing through a

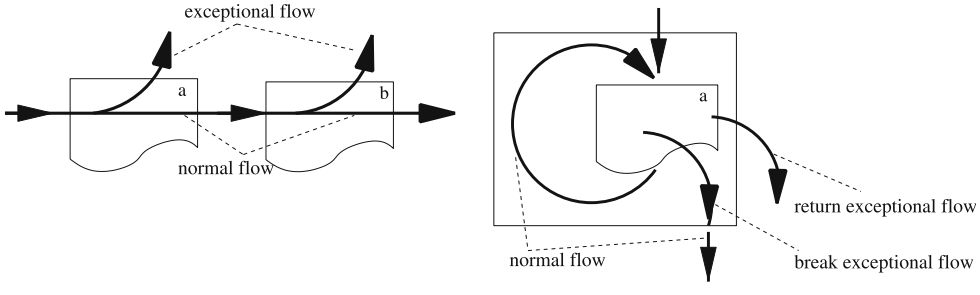

Fig. 3 (*Left*) Normal and exceptional flow through two program fragments in sequence; (*Right*) the exceptional break flow from the body of a forever loop is the normal loop exit.

Table 3 The R and B components of the full logic

$$\frac{p \rhd t_1 \ \text{R: a}\ q \rhd t_2}{p \rhd t_1 \quad \text{R: a;b;}\ q \rhd t_2} \qquad \frac{p \rhd t_1 \ \text{N: a}\ q \rhd t_2 \quad q \rhd t_2 \ \text{R: b}\ r \rhd t_3}{p \rhd t_1 \ \text{R: a;b;}\ r \rhd t_3} \tag{15}$$

$$\frac{p \rhd t_1 \ \text{B: a}\ q \rhd t_2}{p \rhd t_1 \quad \text{B: a;b;}\ q \rhd t_2} \qquad \frac{p \rhd t_1 \ \text{N: a}\ q \rhd t_2 \quad q \rhd t_2 \ \text{B: b}\ r \rhd t_3}{p \rhd t_1 \ \text{B: a;b;}\ r \rhd t_3} \tag{16}$$

$$\frac{p \Rightarrow p' \qquad p' \rhd 1 \quad \text{N: a;} \qquad p' \rhd \bot \qquad p' \rhd 1 \quad \text{R: a;} \qquad q \rhd \bot}{p \rhd t \quad \text{R: while(1) a;} \quad q \rhd \top} \tag{17}$$

$$\frac{p \Rightarrow p' \qquad p' \rhd 1 \quad \text{N: a;} \qquad p' \rhd \bot \qquad p' \rhd 1 \quad \text{B: a;} \qquad q \rhd \bot}{p \rhd t \quad \text{N: while(1) a;} \quad q \rhd \top} \tag{18}$$

$$\frac{\begin{array}{l} p \rhd t \quad \text{N: x} \quad ((x \neq 0) \wedge q_1 \rhd x \ [\!] \ (x = 0) \wedge q_0 \rhd x) \\ (x \neq 0) \wedge q_1 \rhd x \quad \text{R: a;} \quad r_1 \rhd t_1 \qquad (x = 0) \wedge q_0 \rhd x \quad \text{R: b;} \quad r_0 \rhd t_0 \end{array}}{p \rhd t \quad \text{R: if (x) a; else b;} \quad r_1 \vee r_0 \rhd \top} \tag{19}$$

$$\frac{\begin{array}{l} p \rhd t \quad \text{N: x} \quad ((x \neq 0) \wedge q_1 \rhd x \ [\!] \ (x = 0) \wedge q_0 \rhd x) \\ (x \neq 0) \wedge q_1 \rhd x \quad \text{B: a;} \quad r_1 \rhd t_1 \qquad (x = 0) \wedge q_0 \rhd x \quad \text{B: b;} \quad r_0 \rhd t_0 \end{array}}{p \rhd t \quad \text{B: if (x) a; else b;} \quad r_1 \vee r_0 \rhd \top} \tag{20}$$

`spin_lock(&x)` or `spin_unlock(&x)` call respectively increments and decrements the total lock count and the following rules are therefore appropriate:

$$\frac{}{p \rhd \bot \quad \text{N: spin\_lock(xp)} \quad p[\rho - 1/\rho] \rhd \top}$$

and

$$\frac{}{p \rhd \bot \quad \text{N: spin\_unlock(xp)} \quad p[\rho + 1/\rho] \rhd \top}$$

There are no rules giving a break or return semantics, since one cannot break or return from a lock or unlock call. They are atomic (another atomic statement is the empty statement, which also only has an N semantics, since there can be no break or return from it).

Definition 6 *The domain for the NRB interpretation semantics of programs is the set of semantic relations in*

$$(P, T) \leftrightarrow (A(P, T), A(P, T), A(P, T))$$

(*equivalently, triples* $((P, T) \leftrightarrow A(P, T), (P, T) \leftrightarrow A(P, T), (P, T) \leftrightarrow A(P, T))$ *since semantic relations are defined on every possible left argument, cf.* (3)*, and functions* (f_1, f_2, f_3) *map uniquely to the function* $x \mapsto (f_1(x), f_2(x), f_3(x))$*, in which the first component on the right represents the N logic, the second component represents the R logic, and the third the B logic.*

Refinement in this domain is the product of the refinement on the projections. The semantic relations in this domain are those closed with respect to strengthening of the left-hand-side argument and weakening of any of the right-hand-side arguments, being defined for any left-hand-side argument.

Lemma 8 *A weaker logical deduction system for NRB gives a more approximate interpretation.*

Proof This is Lemma 6 restated for the product domain. Again more approximate means a more sparsely populated semantic relation (in any and each of the three components), which means a weaker proof system. □

The (fourth) G component of the logic is responsible for the proper treatment of `goto` statements. To allow this, the whole logic – each of the components N, R, B – works within the additional context of a set e of labelled `goto` conditions. These are predicates that will take effect when the corresponding labelled statement is encountered.

Thus, for example, the full treatment of the normal (N) semantics of sequence is written

$$\frac{e \vdash p \rhd t_1 \ \text{N: a;}\ q \rhd t_2 \qquad e \vdash q \rhd t_2 \ \text{N: b;}\ q \rhd t_3}{e \vdash p \rhd t_1 \ \text{N: a;b;}\ q \rhd t_3}$$

When the analysis gets to a label l, we want to check that the condition q claimed to hold just after l in the context does in fact hold given what we have deduced during the analysis to hold just before the label:

$$\frac{p \Rightarrow q}{\text{l:}q, e \vdash p \rhd t \quad \text{N: l:} \quad q \rhd t}$$

This says that the condition that holds just before the label according to the analysis places a lower bound on the assumption, expressed in the context, about what can hold just after the label. In practice, when we get to such a logical deduction in the analysis, we update the environment with $q' = p \cup q$ and carry on, checking only later on for overall consistency.

Other lower bounds are provided by every `goto l;` that we encounter in the analysis, because these provide alternative routes for getting to the label.

$$\frac{p \Rightarrow q}{\texttt{l}{:}q, e \vdash p \rhd t \quad \text{N:}\,\texttt{goto l;} \quad \text{F} \rhd \top}$$

Note that we cannot run past a `goto` in the normal sequence of execution – the postcondition is F (false). One cannot return or break out of a `goto` either.

If there are backward-going `goto`s in the program being analysed, then the context e is the result of a fixpoint calculation, and, at the very worst, one can use T (true) as the entry in the context for the label. The technique set out in the previous section for finding a fixpoint does apply. However, it requires a multi-pass analysis, and at the present moment our analysis tool is one-pass, so we cannot do that. Instead we currently simply treat forward `goto`s properly and flag backward `goto`s as dangerous if the condition generated in the context (treating it as a forward `goto`) is not already a fixpoint.

If there are only forward-going `goto`s in the program being analysed, then the correct context is calculated by starting with no assumptions and loading the context for label l with the disjunctive union of the conditions discovered at each `goto l;`, plus also the condition that holds just before the label is reached via the normal sequence of execution. Then one can eventually discharge the accumulated union condition as one passes by the label.

7 Logic implementation

The static analyser tool that we have created allows the approximating program logic of C to be configured in detail by the user. The motive was originally to make sure that the logic was implemented in a bug-free way – writing the logic directly in C made for too low-level an implementation for what is a very high-level set of concepts. So a compiler into C for specifications of the program logic was written and incorporated into the analysis tool.

The *logic compiler* understands specifications of the format

```
ctx pre-context, precondition ->
    pre-term :: name(arguments) =
    postconditions with ctx
    post-context -> post-term;
```

where the *precondition* is an input argument, the entry condition for a code fragment, and *postconditions* is an output, a tuple consisting of the N, R, B exit conditions according to the logic. The *pre-context* is the prevailing `goto` context. The *post-context* is the output `goto` context, consisting of a set of labelled conditions. The *pre-term* is the value (term) bound on entry to the program, and the *post-term* is the term bound on *normal* exit from the program. It is a fact that in the event of a break or other exceptional exit from a program, the GNU C compiler carries the void value out with it.

For example, the specification of the empty statement logic is:

```
ctx e, p->t::empty() = (p, F, F)
        with ctx e -> t;
```

signifying that the empty statement preserves the entry condition `p` on normal exit (`p`), and cannot exit via return (`F`) or break (`F`). The context (`e`) and bound term (`t`) are unaltered.

The analysis propagates a specified initial condition forward through the program, developing postconditions after each program statement that are checked for conformity with a specified objective. The full set of logic specifications is given in Table 4. To relate it to the logic presentation in Sect. 6, keep in mind that:

`ctx` e, p `->` t `::` k`()` `=` (n, r, b) `with ctx` e' `->` t'`;`

means

$$\frac{e \vdash \cdots}{e' \vdash p \rhd t \quad \text{N:}\,k \quad n \rhd t'} \qquad \frac{e \vdash \cdots}{e' \vdash p \rhd t \quad \text{R:}\,k \quad r \rhd \top} \qquad \frac{e \vdash \cdots}{e' \vdash p \rhd t \quad \text{B:}\,k \quad b \rhd \top}$$

written out in the notation of Sect. 6. The space above the lines in the rules is filled by the antecedents in those specifications that have `where` clauses in the table.

The undefined term $\bot$ is represented in the table by `NAN`, and the over-defined term $\top$ is represented in the table by `!` (void). Advantage is taken of the fact that the R (return) and B (break) logics always return the term $\top$, so the result value need not be specified per logic – only the N logic returns a significant term value result. The `fix(n,p)` syntax in the specification for a `while` loop means to find a fixpoint above the initial p by increasing p until the n that is calculated comes out below it. The logic of expressions is not represented in this table.

8 Perspectives

One or several objective functions for an analysis are specified by an `objective` specification, presented using the same syntax as that used to specify the approximating logic. Each objective function forms part of a particular *perspective* on the analysis. For the perspective which detects sleep under spinlock, for example, the

Table 4 The program logic of C, as specified to the logic compiler

```
ctx e, p->t::for(stmt)      = (n∨b, r, F) with ctx f -> !
                              where ctx e, p::stmt = (n,r,b) with ctx f;
ctx e, p->t::empty()        = (p, F, F) with ctx e -> t;
ctx e, p->t::unlk(label l)  = (p[n+1/n], F, F) with ctx e -> !;
ctx e, p->t::lock(label l)  = (p[n-1/n], F, F) with ctx e -> !;
ctx e, p->t::assembler()    = (p, F, F) with ctx e -> !;
ctx e, p->t::function()     = (p, F, F) with ctx e -> NAN;
ctx e, p->t::sleep(label l) = (p, F, F) with ctx e -> 0
                              { if (objective(p) ≥ 0) setflags(SLEEP); };
ctx e, p->t::seq(s1, s2)    = (n2, r1 ∨r2, b1 ∨b2) with ctx g -> v
                              where ctx f, n1->u::s2 = (n2,r2,b2) with ctx g -> v
                              and   ctx e, p->t::s1 = (n1,r1,b1) with ctx f -> u;
ctx e, p->t::switch(stmt)   = (n∨b, r, F) with ctx f -> !
                              where ctx e, p->t::stmt = (n,r,b) with ctx f;
ctx e, p->t::if(s1, s2)     = (n1 ∨n2, r1 ∨r2, b1 ∨b2) with ctx f1 ∨f2 -> !
                              where ctx e, p->t::s1 = (n1,r1,b1) with ctx f1
                              and   ctx e, p->t::s2 = (n2,r2,b2) with ctx f2;
ctx e, p->t::while(stmt)    = (n∨b, r, F) with ctx f -> !
                              where ctx e, p->t::stmt = (n,r,b) with ctx f
                              and   fix(n,p);
ctx e, p->t::do(stmt)       = (n∨b, r, F) with ctx f -> !
                              where ctx e, p->t::stmt = (n,r,b) with ctx f
                              and   fix(n,p);
ctx e, p->t::goto(label l)  = (F, F, F) with ctx e∨{l::p} -> t;
ctx e, p->t::continue()     = (F, F, p) with ctx e -> !;
ctx e, p->t::break()        = (F, F, p) with ctx e -> !;
ctx e, p->t::return()       = (F, p, F) with ctx {} -> !;
ctx e, p->t::label(label l) = (p∨e.l, F, F) with ctx e\\l -> t;
```

Legend			
`assembler`	– *gcc inline assembly code;*	`if`	– *C conditional statement;*
`sleep`	– *call to function which can sleep;*	`switch`	– *C case statement;*
`function`	– *call to other C functions;*	`while`	– *C while loop;*
`seq`	– *two statements in sequence;*	`do`	– *C do while loop;*
`NAN`	– *the undefined value;*	`label`	– *labelled statements.*
		`!`	– *the void value.*

corresponding objective function specification is shown in Table 6. There, the significant part in the objective function specification is the term

`upper[n:p]`

which gives the estimated upper limit of the (spinlock) counter `n` subject to the constraints in the state description `p` at that point. The limit is $+\infty$ if `p` is true (`T`). The predicate must bound `n` away from positive values if the objective is not to generate a positive value, and a less strict predicate will cause a more positive value to be calculated as the spinlock count upper bound.

The objective function is computed at each node of the syntax tree. Positive values of the objective function are reported to the user (if the trigger–action rules which will be described in the following part of this section are in force). In particular, calls to functions which can `sleep` at a node where the objective function is positive are reported (this indicates where a call to a sleepy function might occur under spinlock).

There is also an initial state description specified to the logic compiler, also set out in Table 6. For a perspective that checks for sleep under spinlock, the initial proposition that is asserted is:

`(n` $\leq$ `0)`

It says here that the spinlock counter `n` is less than or equal to zero (actually, exactly zero is intended, but the inequality is just as good and simpler to compute).

Logic propagation through the syntax tree of a program source code is complemented by a trigger–action system which acts whenever a property changes at a node. For the perspective which detects sleep under spinlock, the rules in Table 5 are applied. Their principal aim is to construct the list of sleepy functions, checking for calls by name of already known sleepy functions and thus constructing the transitive closure of the list under the call (by name) graph.

Rule (1) applies whenever a function is newly marked as sleepy (`SLEEP!`). Then if the `objective` function (here the maximal value of the spinlock count `n`) has already been calculated on that node (`OBJECTIVE_SET`) and is not negative (`OBJECTIVE` ≥ 0, indicating that the spinlock count is 0 or higher) then all the known *aliases*

Table 5 Defining initial conditions, and an objective function to be calculated at every node of the syntax tree.

```
::initial()   = (n≤0);
p::objective() = upper[n:p];
```

(other syntactic nodes which refer to the same semantic entity) are also marked sleepy, as are all the known *callers* (by name) of this node (which will be the current surrounding function, plus all callers of aliases of this node).

The reason why sleepiness is not propagated under *negative* spinlock is quite subtle. Consider function *f* called from function *g* called from function *h*. If the spinlock count is negative at the call of *f* in *g*, then *g* is intended to be called under spinlock (releasing an already released spinlock is a design error). If *f* is sleepy then *g* would ordinarily be marked sleepy too and that would be marked as an error when *g* is called under spinlock in *h*. But that is wrong when *f* is under negative spinlock in *g*, because then *f* is not under spinlock when *g* is called under spinlock in *h* and it is not a problem in *h* that *f* chooses to sleep inside *g*. So, under these conditions, *g* should not be marked as sleepy.

Rule (2) in Table 5 is triggered when a known sleepy function is referenced (`REF!`). Then all the callers (including the new referrer) are marked as sleepy if they were not so-marked before. The `REF` flag is removed as soon as it is added so every new reference triggers this rule. The effect of rules (1) and (2) together is to efficiently create the transitive sleepy call (by name) graph.

A list of all calls to functions that may sleep under a positive spinlock count is created via rule (3) in Table 5. Entries are added when a call is (a) sleepy, (b) the spinlock count at that node is already known, and (c) is non-negative (positive counts will be starred in the output list, but all calls will be listed).

9 More targets

Spinlock-under-spinlock can be detected by first constructing the transitive graph of functions which call functions which take spinlocks, and sounding the alarm at a call of such a function under spinlock.

Making that graph requires attaching the code

```
setflags(SPINLOCK)
```

into the logic of the spin lock function calls in Table 4, just as in the case of the sleep function call logic specification. The trigger–action rules in Table 5 are then duplicated, substituting `SPINLOCK` for `SLEEP` in the existing rules, so that the rules propagate the `SPINLOCK` flag as well as the `SLEEP` flag from called to caller. Then a single trigger–action rule is added which outputs an alert when a function marked with `SPINLOCK` (i.e. a function which calls a function which …takes a spinlock) is called under spinlock:

```
(SPINLOCK & SPIN_SET & SPIN > 0)! → output()
```

files checked:	1151	
alarms raised:	426	(30/1151 files)
false positives:	214/426	
real errors:	212/426	(3/1151 files)
time taken:	~6h	
LoC:	650K	(unexpanded)

Fig. 4 Testing for access to kfreed memory in the 2.6.3 Linux kernel

Why is taking a spinlock twice dangerous? Taking the same spinlock twice is deadly, as Linux kernel spinlocks are not reentrant. The result will be to send the CPU into a busy forever loop. Taking two different spinlocks one under the other in the same thread is not dangerous, unless another thread takes the same two spinlocks, one under the other, in the reverse order. There is a short window where both threads can take one spinlock and then busy-wait for the other thread to release the spinlock they have not yet taken, thus spinning both CPUs simultaneously and blocking further process. In general, there is a deadlock window like this if there exists any spinlock cycle such that A is taken under B, B is taken under C, etc. Detecting double-takes flags the potential danger.

We have also been able to detect *accesses to freed memory* (including frees of freed memory). The technique consists of setting the logic of a `kfree` call on a variable containing a memory address to increment a counter variable `a(l)` unique to the variable. The (integer index label `l` generated by the analysis) Assigning the variable again resets the counter to zero (`p[!a(l)]` means proposition `p` relaxed to remove references to the counter `a(l)`; `a` is treated like a vector where appropriate, so initial condition `a` ≤ `0` has `a(l)` ≤ `0` too):

```
ctx e, p ::kfree(label l)
= (p[a(l)-1/a(l)], F, F) with ctx e;
ctx e, p ::assignment(label l)
= (p[!a(l)], F, F) with ctx e;
```

The alarm is sounded when the symbol with label `l` is accessed where the counter `a(l)` may take a positive value – a variable with index `l` may point to freed memory.

A survey of 1,151 C source files in the Linux 2.6.3 kernel reported 426 alarms but most of these were clusters with a single origin. Exactly 30 of the 1,151 files

Table 6 Trigger–action rules which propagate information through the syntax tree.

```
1.  SLEEP! & OBJECTIVE_SET & OBJECTIVE ≥ 0   → aliases |= SLEEP,
                                               callers |= SLEEP
2.  REF! & SLEEP                            → callers |= SLEEP, ~REF
3.  (SLEEP & OBJECTIVE_SET & OBJECTIVE ≥ 0)! → output()
```

Fig. 5 Access to kfreed memory in kernel 2.6.3

File & function	Code fragment	
fm801-gp.c: fm801_gp_probe	101	kfree(gp);
	102	printk("unable to grab region 0x%x-0x%x\n", gp->gameport.io, gp->gameport.io + 0x0f);
aic7xxx_old.c: aic7xxx_detect	9240	while(current_p && temp_p)
	9241	{
	9242	if (((current_p->pci_bus==temp_p->pci_bus)&&...){
		...
	9248	kfree(temp_p);
	9249	continue;

were reported as suspicious in total (see Fig. 4). One of these (`aic7xxx_old.c`) generated 209 of the alarms, another (`aic7xxx_proc.c`) 80, another (`cpqphp_ctrl.c`) 54, another 23, another 10, then 8, 7, 5, 4, 2, 2, 2, 2, and the rest 1 alarm each. Three of the flagged files contained real errors of the type searched for. Two of the error regions are shown in Fig. 5. Curiously, `drivers/scsi/aic7xxx_old.c` is flagged correctly, as can be seen in the second code segment in the figure.

All the false alarms were due to a bug in the postcondition logic of assignment at the time of the experiment, which caused a new assignment to x closely following on the heels of a `kfree(`x`)` to be (erroneously) flagged.

A repeat experiment on 1,646 source files (982,000 lines, unexpanded) of the Linux 2.6.12.3 kernel found that all the errors detected in the experiment on kernel 2.6.3 had been repaired, and no further errors were detected. There were eight false alarms given on seven files (all due to a parser bug at the time which led to a field dereference being treated like reference to a variable of the same name).

10 Software

The source code of the software described in this article is available for download from `ftp://oboe.it.uc3m.es/pub/Programs/c-1.2.13.tgz` under the conditions of the GNU public licence (GPL), version 2.

11 Summary

The notion of symbolic approximation has been introduced in order to describe the working of a practical C source static analyser, initially aimed at the Linux kernel source. The analyser is capable of dealing with the millions of lines of code in the kernel source on a reasonable time scale, at a few seconds per file. The approximating logic is configured by an expert to obtain different analyses which an unskilled user can then apply (and several analyses are performed at once). Symbolic approximation constructs a representation of the program in a symbolic domain where a precise meaning for approximation has been defined which says that less is deducible about a more approximate representation. Thus making the analysis logic less powerful makes what it says about the program into statements about a more approximate representation of the program, a tradeoff that can be exploited in order to improve efficiency.

The particular logical analyses described here (perspectives) have detected about two uncorrected deadlock situations per thousand files in the Linux 2.6 kernel, and about three per thousand files which access already freed memory.

Acknowledgements This work has been partly supported by funding from the EVERYWARE (MCyT No. TIC2003-08995-C02-01) project, to which we express our thanks.

References

1. Ball T, Rajamani SK (2002) The SLAM project: debugging system software via static analysis. In: Proc. POPL '02: Proceedings of the ACM SIGPLAN-SIGACT conference on principles of programming languages
2. Breuer PT, Bowen JP (1995) A PREttier compiler–compiler: Generating higher order parsers in C. Softw Pract Exp 25(11):1263–1297
3. Breuer PT, Pickin S (2006) One million (Loc) and counting: static analysis for errors and vulnerabilites in the Linux kernel

source code. In: Pinho LM, Harbour MG (eds) Proceedings of Reliable Software Technologies—Ada-Europe 2006, 11th Ada-Europe international conference on Reliable Software Technologies, Ser. LNCS, PP. 56–70
4. Breuer PT, Garcia Valls M (2004) Static deadlock detection in the Linux kernel. In: Llamosí A, Strohmeier A (eds.) Reliable software technologies – Ada-Europe 2004, 9th Ada-Europe International Conference on Reliable Software Technologies, Palma de Mallorca, Spain, 14–18, June 2004. LNCS vol 3063 Springer, Berlin Heidelberg New York, pp 52–64.
5. Breuer PT, Delgado Kloos C, Martínez Madrid N, López Marin A, Sánchez L (1997) A refinement calculus for the synthesis of verified digital or analog hardware descriptions in VHDL. ACM Trans Program Lang Syst (TOPLAS) 19(4):586–616
6. Breuer PT, Martínez Madrid N, Sánchez L, Marín A, Delgado Kloos C (1996) A formal method for specification and refinement of real-time systems. In: Proceedings of the 8th EuroMicro workshop on real time systems, IEEE, New York, L'aquilla, Italy, pp. 34–42
7. Chaki S, Clarke E, Groce A, Jha S, Veith H (2003) Modular verification of software components in C. In: Proceedings of the international conference on software engineering, pp. 385–389
8. Clarke E, Emerson E, Sistla A (1986) Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. ACM Trans Program alang Syst 16(5):1512–1542
9. Clarke E, Grumberg O, Long D.A (1994) Model Checking and Abstraction. ACM Trans Program Lang Syst 16(5):1512–1542
10. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM symposium on the principles of programming languages, pp 238–252
11. Chen H, Dean D, Wagner D (2004) Model checking one million lines of C code. In: Proceedings of the 11th annual network and distributed system security symposium, San Diego, CA
12. Foster JS, Fähndrich M, Aiken A (1999) A theory of type qualifiers. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI'99), Atlanta, Georgia
13. Foster JS, Terauchi T, Aiken A (2002) Flow-sensitive type qualifiers. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI'02), Berlin, Germany, pp 1–12
14. Johnson R, Wagner D (2004) Finding User/Kernel pointer bugs with type inference. In: Proceedings of the 13th USENIX security symposium, 9–13 August 2004, San Diego, CA, USA
15. Wagner D, Foster JS, Brewer EA, Aiken A (2000) A first step towards automated detection of buffer overrun vulnerabilities. In: Proceedings of the network and distributed system security (NDSS) symposium, 2–4 February 2000, San Diego, CA, USA