University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Cooperatively combining program verifiers: foundations and tool support

Nathaniel Charlton

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, August 2008

Declaration

I declare that the work presented in this thesis is my own, conducted under the supervision and guidance of Dr. Michael Huth, and that no part it has been submitted for any other academic award.

Abstract

Computer science literature abounds with descriptions of program verifiers, systems which analyse a software program and attempt to prove automatically that the program satisfies behavioural specifications. Techniques used include predicate abstraction, three-valued heaps graphs and classes of polyhedra. Yet while these systems have had some encouraging successes, each deals only with particular patterns of program behaviour: e.g. predicate abstraction can infer arithmetical relationships but does not capture the "shape" of linked data structures; for three-valued shape graphs the reverse is true. Thus typical programs, in which different patterns of behaviour are mixed up together, still cannot be verified automatically.

This thesis explores the question: "By combining several program verifiers, and making them cooperate, can we produce a verification system that solves a broader range of verification problems than its components do?". Specifically, our approach is to allow the verifiers to exchange information about program states, expressed as formulae of a single common logic, so that each can benefit from the others' findings.

We design a mechanism which enables the verifiers to cooperate. Our setup comprises several verifiers, cleanly separated as *analysis modules* implementing a common interface, and a central "broker" which oversees the verification process, propagating formulae between the analysis modules. We formalise this approach for programs of a core imperative heap-manipulating language with recursion, annotated with correctness assertions. We give an interprocedural verification algorithm for the broker and soundness conditions for analysis modules, and prove that these ensure the algorithm is sound (though perforce incomplete).

We report on the implementation of our new method in an experimental system HECTOR, which includes the broker and analysis modules for a range of techniques, including predicate abstraction and three-valued shape analysis. By means of a verification case study, we demonstrate some of the advantages of our approach.

Acknowledgements

I would first of all like to thank my supervisor Michael Huth, whose unfaltering optimism and encouragement enabled me to keep working even when I felt thoroughly disconsolate. I wish I had been better able to listen to his good advice.

I thank everyone with whom I have discussed aspects of this research, particularly Ian Hodkinson and Philippa Gardner, and the members of her research group. I would also like to thank Joe Stoy, my former tutor, whose excellent teaching gave my computer science education such a good start.

I wish to express my personal gratitude to all of my friends for their kindness and support. Warm thanks are due in particular to my partner Oi Tak. I am extremely fortunate to have met my office-mates Jaspreet, Jayshan, Alex and Simon, with whom I have shared the extraordinary PhD experience.

Finally, I am eternally grateful to my parents, to whom I owe so much, for allowing me the freedom to set my own goals in life, and always supporting me in the pursuit of them.

Dedicated to the memory of Tupac. Rest in peace.

The tropical sunshine lay like warm honey on the naked bodies of children tumbling promiscuously among the hibiscus blossoms. Home was in any one of twenty palm-thatched houses. In the Trobriands conception was the work of ancestral ghosts; nobody had ever heard of a father.

Aldous Huxley, Brave New World

Contents

List of Figures

Chapter 1

Introduction

1.1 Setting the scene

Over the last few decades, more and more tasks occurring in our daily lives have been turned over to computer software. The size and complexity of programs is increasing all the time, in line with advances in hardware and programming paradigms. Yet the problem of software reliability remains: software continues to be faulty, and because of greater dependence on software, these faults are costlier than ever. The failure of safety-critical control systems may cost human lives, and businesses stand to lose huge amounts of money if products are defective. Thus, there is an urgent need to address the reliability problem.

The idea of automatic software verification is to turn the task of ensuring software correctness over to software, just as we have done with so many other tasks. We would like to build a system which takes as input a program, plus a formal description of how that program should behave, and produces as output either confirmation that the program has the desired behaviour, or an explanation of why it does not (such as an execution trace showing the undesired behaviour).

Conventional testing, while having an important role in the development process,

does not answer the verification problem because, as Dijkstra famously remarked, "program testing can be used to show the presence of bugs, but never to show their absence" [Dij70]. Put another way, testing does not give strong enough assurances of correctness. Something else is required.

1.1.1 Verification of finite-state systems

For systems which are finite state, such as many control systems and communication protocols, the *model checking* technique [CE82, QS82] is widely used to verify temporal properties such as deadlock-freedom. In model checking, the system's state space is explored exhaustively, resulting in a full verification: if the system satisfies the property then this is reported, and if it does not, a counterexample execution is given. In other words, model checking is a *decision procedure* for finite state verification problems. Efficient model checking algorithms based on BDDs (binary decision diagrams) can now handle systems with upwards of 10^{20} states [BCM+92], and the technology has established itself in industry e.g. [GL00, RL02]. The model checking process does not require any human intervention, and most model checkers allow a wide range of behavioural properties to be checked.

1.1.2 Verification of infinite-state software

Most applications programs, however, are infinite state. "Real" programming languages such as Java contain numerous constructs which may give rise to an infinite state space: for example, a true integer type, arbitrary-depth recursion, arbitrary-length arrays, and heap allocation which allows for instance arbitrary-length linked lists.

Unfortunately[1], the situation for such infinite state systems is not as rosy. The

[1]Strictly speaking, I am unsure how much sense it makes to describe as "unfortunate" a situation which seems to be logically entirely inevitable; philosophy isn't our aim here, however.

exhaustive state checks used in finite state model checking are in general no longer possible. In fact, as we review in the next chapter, any suitably general verification problem for infinite state software is undecidable. This means that we cannot design a "perfect" program verifier which, running fully automatically and for any input program, will always terminate, never says "don't know" and is always accurate about whether the program satisfies its specification.

Nevertheless, the software reliability problem does not go away simply because of a formal result; we cannot just throw up our hands and go home. So some of the attributes of the "perfect" verifier (full automation, full range of input programs, guaranteed termination, guaranteed definite result, and guaranteed accuracy) must be sacrificed. On the other hand, what gives us hope is that the range of programs that programmers would ever actually write is a tiny fraction of the theoretical range, consisting of the same basic idioms applied over and over. In particular, the programs used in practice generate states which are in some sense orderly, regular and full of patterns.

In approaches to verifying such software, the infiniteness of the state space is overcome by the process of *abstraction*: we define some *symbolic* way to represent not states, but entire *sets* of states (possibly infinite). Operations on these sets, such as computing successors under a program instruction, or testing their emptiness, are performed by symbolic reasoning algorithms. This basic method surfaces in different guises, such as *abstract model checking* [CGL92], *abstract interpretation* [CC77, CC92a] and *Hoare logic proofs* [Hoa69]. Using these symbolic representations one then builds, manually or automatically, a finite representation of the program, sometimes called an *abstract model*, which can be checked for possible errors.

Of course, the original program and the abstract model must satisfy an appropriate relation to ensure that the results obtained from checking the abstract model will "carry over" to the original program. For example, we may construct the abstract model to be a *simulation* of the original program (e.g. [BG03]), which means that

all execution paths in the program are possible in the abstract model (and possibly more). This ensures that safety properties which hold in the abstract model also hold in the program. Alternatively, we may think of this process in terms of computing an inductive invariant for the program.

Checking software in this way has achieved some success: for example, communication protocols, garbage collectors and libraries implementing data structures have been verified in this manner (e.g. [HS96, DDP99, MS01] respectively). However, as we shall see shortly, current techniques have significant limitations.

1.1.3 The choice of abstraction domain

A crucial question when applying abstraction is: what class of symbolic representations should we use? Or, put another way, what should be our *abstract domain*? Some well-known representations, or abstract domains, are:

intervals: for each program variable x an interval is computed, i.e. a constraint $a \leq x \leq b$ where $a \in \{-\infty\} \cup \mathbb{Z}$ and $b \in \{+\infty\} \cup \mathbb{Z}$ (as in [CC76, SW04])

polyhedra: the possible values of numerical variables are represented by a convex polyhedron (e.g. [CH78, CC04])

three-valued shape graphs: the contents of the heap is represented imprecisely but safely by a three-valued heap [SRW99, LAMS04] (e.g. Figure 2.10)

graph types: the layout of the heap is represented by a *graph type declaration* [MS01] (e.g. Figure 2.8)

monomials: in monomial predicate abstraction [GS97, BPR01] the concrete program states are grouped into equivalence classes based on the values they give to a finite collection of first order predicates

buffer domains: these provide a specialised representation to track the use of string buffers in C [SK02]

The choice of abstraction domain affects most characteristics of the verification system, such as how much time it takes to run, whether it is fully automatic or needs to be guided by user annotations, how precise the analysis is and, perhaps most importantly, which subclasses of programs and properties can be verified. For example:

- Interval analysis is fast and simple and fully automatic, and each constraint needs little space; the information generated is very limited, but may be sufficient if we need only check against out-of-bounds array accesses or integer overflows.
- Polyhedral analysis discovers slightly more information, in that it can find relationships between variables, but has a greater computational cost.
- Predicate abstraction can find much more general relationships between variables, and has been used to successfully verify interface usage properties of device drivers [BBC+06]. However, the approach is harder to use without user guidance (because the abstraction predicates must somehow be chosen), may have even higher computational cost (because successor computations require invocation of a theorem prover), and does not handle heap-manipulating programs well (because first order predicates cannot express many important heap properties).
- Conversely, graph types and three-valued shape graphs do handle heap properties well, but cannot track arithmetical relationships (at least not without *ad-hoc* contortions). Graph types give a very precise analysis of the heap, but are only applicable to a subset of data structures; three-valued shape graphs apply to any heap, but sometimes give the answer "don't know".

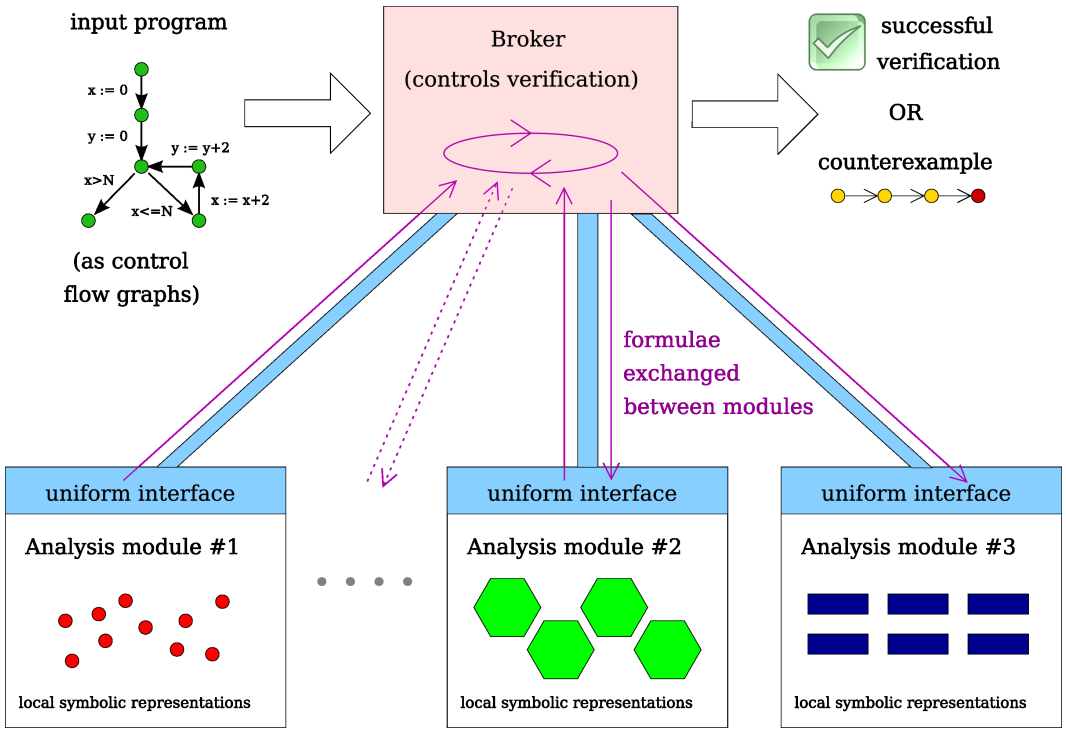

Figure 1.1: The overall structure of our approach and prototype verifier

Another domain one could add into the above list is:

full predicate logic: any formula of an appropriate predicate logic can be used to describe a set of states (as in [Hoa69] and [Dij75])

although systems built using full predicate logic (e.g. [SI99, IEI04]) are of a somewhat different character to those built using more restricted domains.

1.2 Our idea: clean cooperation between domains

1.2.1 The overall design of our system

In this work we propose a system in which the various abstraction domains can be used cooperatively to analyse the target program.

Our setup, as shown in Figure 1.1, consists of several *analysis modules*, one per

abstraction domain, and a central "broker" which overseas the verification. The abstraction domains we wish to use are wrapped inside analysis modules, which implement a common interface. This separation is clean in the sense that the symbolic representations used for a domain never escape outside its analysis module, and the broker and other modules know nothing about them.

Crucially, the interface allows each module to share information about program states, expressed using a single common logic; a module may make facts about program states available to the other modules, and conversely is able to receive such facts. In this way the modules can cooperate, using each others' information to make their own analysis more precise. The broker coordinates the propagation of formulae between the modules. As our single common logic we have chosen a first order logic with transitive closure.

1.2.2 Motivation: conjectured advantages of our system

It is well known that the symbolic representations used in each abstraction domain are in general not independent, and therefore the domains can benefit from each others' information, using it to make their own analysis more precise. When beginning this work, we could see two ways in which this might be beneficial:

B1 Verifying programs with diverse features:

We saw in the previous section that the abstract domains in use target specific aspects of program behaviour, such as numerical relationships, linked data structures, use of string buffers etc.; doing so brings increased efficiency and automation and makes the design of the domains more manageable. But in applications programs, these features are all mixed up together: for instance a program may use both linked data structures and buffers to store data which is processed numerically. Verification systems based on a single specialised domain cannot verify such programs. By combining the different domains

cooperatively, we hoped our system would solve a broader range of verification problems than its components do.

B2 Using cheap analyses where applicable reduces workload:

We suspected that by using cheap but shallow analyses alongside more expensive but deeper ones, we could reduce the workload of the more expensive analyses, because 1. "easy cases" would be taken care of entirely by the cheaper analyses, and 2. for "difficult cases" the cheaper analyses might still supply some helpful information to the more expensive ones. That is, we hoped to gain the depth of the more expensive analyses, but with less computation.

For example, a cheap shallow constant propagation analysis may reduce the work of predicate abstraction, because abstraction predicates tracking the constant values are no longer needed, so there are fewer calls to the theorem prover.

The key point about our system is that we try to take advantage of the non-independence of domains in a modular way. By placing each abstraction domain behind a common interface, we aimed for the following advantages:

B3 Implementation is more manageable:

The implementor of an analysis module just needs to implement the module interface: he only has to make his new module "understand" the single common logic, and the new module will then cooperate with existing ones. The implementor does not need to know about the abstract constraints used internally by other modules. Thus, we can develop the verification system in small, easy to understand parts.

B4 Abstractions can be mixed and matched, with the most appropriate chosen for each task:

Different verification tasks require or benefit from different kinds of abstractions. In our scheme, once abstractions have been suitably wrapped as modules, they can be "mixed and matched" freely, so we can select whatever kinds are best for each task.

An alternative approach would be to program the interactions between each pair of domains directly, rather than going through a broker and common logic. We suspect that this would bring benefits of flexibility and speed: constraints could be propagated in whatever form is most convenient, and in fact the different analyses need not even explore the program according to the same schedule; they could be coroutined in arbitrary ways. But it would also have disadvantages: the interactions between domains can be subtle and hence difficult and time-consuming to implement directly, and the implementor needs to understand the structure of all the domains involved. Additionally, a direct implementation would have to be redone each time the set of abstractions used was changed.

1.3 Specific objectives of this thesis

The conjectured benefits just described are long-term goals for the approach of modular domain combination. As such, they are very ambitious, and (necessarily) a bit nebulous. This doesn't detract from their importance, but means that we needed to set some more immediate goals for this work; "a journey of a thousand miles begins under one's feet".

Thus the plan was to design and build a prototype verification system based on our idea for modular domain combination, and experiment with using it to verify some example programs, including a moderately-sized case study. We hoped that these experiments would show enough encouraging results to motivate and inform further investigation of the approach.

In particular, for our case study, we hoped that the verification would have the following characteristics:

E1 The program verified is moderately sized, is based on a piece of real-world software, and uses diverse features.

E2 The verification employs (at least) two sophisticated domains, which are significantly different.

E3 There is a *two-way* exchange of information between the sophisticated domains: each contributes information which the other uses to make its own analysis better in interesting ways.

E4 One or more additional domains implementing shallow analyses are shown to help out the sophisticated domains, by handling easy cases or supplying information that can be easily obtained. The verification should run *faster* when these additional domains are used.

E5 The domains used should implement techniques that are useful for software verification in general, and not be developed specifically just to make the case study work.

With these objectives in mind, we chose for our sophisticated domains the predicate abstraction and three-valued shape analysis techniques, for the following reasons. These techniques work in fundamentally different ways (the latter is model-based, while the former uses theorem proving) and have significantly different uses (the latter is good for modelling linked heap structures but ignores numerical operations, while the former accounts for numerical operations but doesn't handle linked heap structures well). Predicate abstraction is widely known and successful; three-valued shape analysis is also well-known and extensively written about. Both are widely applicable techniques and are supported by existing freely available software, such

as the Simplify theorem prover [DNS05] and TVLA [SRW99, LAMS04] which we planned to reuse as part of our implementation.

The design and implementation of the prototype verification system was thus to include the following Design and Implementation tasks:

D1 Fix the "target programming language", i.e. the language in which the programs to be verified will be written, and formalise its syntax and semantics. The language should be simple and idealised, but powerful enough to express interesting programs in a straightforward way.

D2 Define the single common language which analysis modules will use to exchange information.

D3 Think about the ways in which the analysis modules need to interact with the central broker, and devise an appropriate interface for the analysis modules to implement.

This includes not only working out the signatures of the functions exported by the interface, but also stating how an analysis module should behave in order to return "sound" results.

D4 Formulate a generic verification algorithm for the broker, which works with whatever set of analysis modules is presented, and propagates information between them so that they cooperate. The algorithm must be proved to produce sound results.

I1 Implement the central broker which coordinates the verification process, using the algorithm from D4.

I2 Implement an analysis module for predicate abstraction, which supports both the trivector and monomial variants.

I3 Develop a module for three-valued shape analysis, by appropriately wrapping the TVLA software. One difficulty here lies in how to "translate" shared

information into the setting of shape graphs, and in the other direction how to extract logical formulae from shape graphs.

I4 Provide an interface by which users can enter programs, configure and start the verification process, and monitor the results, including facilities to inspect the abstract models built, and see where in the model the exchange of information has proved useful and where it has not.

I5 Implement some shallow analysis modules to help out the sophisticated ones.

Of course, this was only an approximate plan, and we fully expected unforeseen issues to arise once we started running and experimenting with the prototype; that was an important purpose of the research.

1.4 Contents of this thesis

Chapter 2: Background draws together the background material to our work. We discuss the verification problem and introduce inductive verification (which appears in different guises such as abstract model checking, abstract interpretation and Hoare logic proofs) in a way that is parametric in the abstraction domain used. We give example verifications using first order formulae and intervals respectively as the domain. We survey a range of abstraction domains put forward in the literature. We end with an example demonstrating that the various abstraction domains are not independent, so that domains can each make their own analysis more precise if they cooperate with each other.

Chapter 3: Our approach: basic concepts and algorithms presents and formalises the fundamentals of our new verification method. We fix and formalise a programming language for our verification methods to target (an idealised imperative heap-manipulating language with recursive procedures). We present a logic over program states (we choose a first order logic extended with transitive closure) which

will serve as the single common language in which our analysis modules exchange information. We then set out our notion of an *analysis module*, which is central to our work: each such module implements an abstraction domain behind a common interface, through which it interacts with the central coordinating "broker", sharing and receiving formulae from other modules. Finally we give a module-based verification algorithm for the broker, which works with any collection of analysis module that is provided (this algorithm is worklist-based and uses procedure summarisation to handle recursion). We prove that the algorithm terminates and produces sound results, provided that all analysis modules used satisfy some stated soundness conditions.

Chapter 4: Implementation: the HECTOR system describes our experimental software tool HECTOR which implements the verification framework developed in Chapter 3, beginning with an overall summary of the implementation. We outline the seven analysis modules implemented in HECTOR: three of these provide sophisticated domains (trivector predicate abstraction, monomial predicate abstraction and three-valued shape graphs) while the other four modules are based on "lightweight" techniques (a basic type system, a sign analysis, symbolic propagation and a rudimentary heap reference tracking domain). As part of this, we discuss how each module provides shared information for other modules, and conversely how each module takes advantage of shared information it receives. Finally we explain how a user interacts with HECTOR, showing some of the customisable graphical output with which the system displays its models and counterexamples.

Chapter 5: Case study reports on our verification of an implementation of the popular puzzle game MineSweeper, which makes use of linked data structures, pointer arithmetic and recursion. This program can be verified neither by conventional predicate abstraction (which cannot handle linked structures) nor by TVLA shape analysis (which doesn't handle pointer arithmetic). Using HECTOR, however, our predicate abstraction and shape analysis modules work cooperatively, and

hence we can verify the program; this is a good success for our approach. To illuminate this, we trace through the model extraction algorithm following a particular "scenario" of MineSweeper gameplay, and pick out interesting instances of formula sharing. Another success for HECTOR is that when we also use our four lightweight modules alongside those for predicate abstraction and shape analysis, the time taken to verify the program significantly decreases, as does the amount of user guidance needed.

Chapter 6: Additional model checking features describes three extensions to our basic verifier. Firstly, we provide a method for checking temporal properties expressed in a "two-level" fragment of LTL; by "two-level" we mean that the "propositions" of the temporal formulae are themselves first-order formulae describing the program's state. Secondly we provide a method for post-pruning the models, that is, cheaply pruning away states from which execution "fizzles out"; such states are automatically infeasible. These extensions were developed because we found that, even with HECTOR's existing customisable graphical output, it became difficult to understand and navigate around models once they had several thousand states. The third extension is a way to support the falsification of safety properties as well as their verification, while *still only performing over-approximation*, by exploiting the seriality implicit in our programs' concrete semantics.

Chapter 7: Conclusions concludes the thesis. We revisit the objectives stated in this introduction and reflect on the extent to which they have been met. We compare our work with other recent approaches to domain combination. Finally, we formulate what we believe to be the most important directions for the future continuation of this research.

1.4.1 Publications

Some material presented in this thesis first appeared in other publications, as follows.

- The workshop paper [Cha06c] first explained the idea of verification based on cooperating analysis modules (or "analysis plugins" as they were then called). The paper focused on motivating the approach and sketching an early version of the verification framework in Chapter 3.

- The journal paper [Cha06a] (and the associated technical report [Cha06b]) detailed, in a far more formal way, a more mature version of the verification framework (though this had still not reached the current form set out in Chapter 3). An early version of (what became) the HECTOR system was described, along with detailed accounts of the analysis modules based on TVLA and on a simple type system.

- The conference "tool paper" [CH07], jointly written with Michael Huth, described the HECTOR implementation from Chapter 4, and outlined the "two-level" temporal logic checking from Chapter 6.

- The workshop paper [CH08], also jointly written with Michael Huth, contained a formal account of the new falsification method discussed in Chapter 6, along with an illustrative example of its use and some additional related material and theorems.

Chapter 2

Background

In this chapter we outline the basic concepts on which our work builds, so that the reader can understand our work and see it in its proper context. As such, here we refer mainly to research published before work on this thesis began. In Chapter 7 we will discuss newer research which was published during the completion of this thesis.

We begin by discussing the verification problem, and how programs and properties are represented. We then explain the idea of inductive verification (which appears in different guises e.g. abstract model checking, abstract interpretation and Hoare logic proofs) in a way that is parametric in the abstraction domain used. Next we survey a range of abstraction domains put forward in the literature, noting how they fit into the general framework of inductive verification. We end with the observation that different abstraction domains are not independent, so if multiple domains cooperate they can each make their own analysis more precise.

```
procedure IntSqrt(N)
  {
    var x;

    x := 0;

    while ((x+1)*(x+1) <= N)
      {
        x := x+1;
      }

    return x;
  }
```

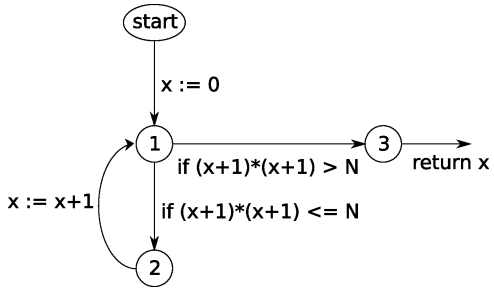

Figure 2.1: An example procedure, for calculating integer square root. The left part shows the procedure as it might be represented in source code; the right part shows the control flow graph which we work with.

2.1 The verification problem

As stated in the introduction, the verification problem is: given a program and a description of its desired properties as input, attempt to determine whether or not the program does indeed have the desired properties.

2.1.1 Programs as control flow graphs

Programs are written as source code, as in Figure 2.1 (left): the procedure given is for calculating integer square roots. But while source code is a good input format for programmers, it is inconvenient for program analysis tools. For this reason, many such tools (e.g. [BR00, LAMS04]) assume that the input programs are represented with *control flow graphs* (CFGs) as in Figure 2.1 (right). We shall take this approach too. We will assume a separate CFG for each procedure in the program. The nodes of a CFG represent control locations in the procedure, and the edges are labelled with atomic statements. Execution of a procedure begins at the node labelled "start" and flows along the arrows.

The problem of converting source code to CFGs needn't worry us because spe-

cialised software packages such as SOOT [VRHS+99] have been written specifically to provide program analysis tools with a convenient representation of programs, and "insulate" the tools from the nuisances of source code. When we build our verification system in Chapter 4 we will assume that such preprocessing has already been done, and will accept as input a textual representation of the CFGs' edges.

A standard way of giving semantics to such a language is to define

1. a set S of *program states*
2. a subset $I \subseteq S$ designated as *initial states*
3. a family of *transfer functions* $f(i) : S \to \mathbb{P}(S)$ indexed by the atomic statements i

Typically for simple programs the state consists of an *environment*, which maps variables to their values; for heap-manipulating programs there is also a *heap* component which maps memory addresses to their contents. Each transfer function $f(i)$ describes the effect of the statement i.

For example, for the program in Figure 2.1, a state is simply a value for N and x, i.e. $S = \mathbb{Z} \times \mathbb{Z}$. The transfer functions for the statements `x := 0`, `x := x+1` and `if (x+1)*(x+1) > N` respectively are

$$f(x := 0)(N, x) = \{(N, 0)\}$$

$$f(x := x + 1)(N, x) = \{(N, x + 1)\}$$

$$f(\text{if } (x+1) * (x+1) > N)(N, x) = \begin{cases} \{(N, x)\} & \text{if } (x+1) * (x+1) > N \\ \emptyset & \text{otherwise} \end{cases}$$

The last of these shows one reason there is a power set in the signature of the $f(i)$s: so that conditional edges can have no successors if their condition is not met.

Another reason is to account for nondeterministic statements, such as a memory allocation command which allocates any available address.

We will use the name *located state* to refer to a pair (l, s) where $s \in S$ and l is a node from the CFG. We will write the set of located states as S_{loc}. For each node of the CFG, we lift the transfer function to a (partial) function $S_{loc} \to \mathbb{P}(S_{loc})$. For node 2 in our example, this lifted function is $(2, (N, x)) \mapsto \{(1, (N, x + 1))\}$. When we do this for each node, and take the union of the resulting relations, we get a transition relation $\mathit{semantics}(P) \subseteq S_{loc} \times S_{loc}$ giving the semantics of the program P.

Finally, we define the set of *reachable* located states as those which are reachable from an initial state by following a number of "steps" in $\mathit{semantics}(P)$. Formally this is defined as a least fixed point:

$$\mathit{reach}(P) = \text{ lfp } X \,.\, (\{\mathit{start}\} \times I) \;\cup\; X \;\cup\; \mathit{semantics}(P)(X)$$

2.1.2 Correctness properties and reachability

Now that we have said what our programs look like, we need a way to specify what properties we would like them to have.

A simple and common way of specifying program properties is to write *preconditions*, *postconditions* and *assertions* (these are found e.g. in the language Eiffel [Mey92], the JML specification language [LBR06] and the Spec# system [BLS05]). A precondition is a condition that should hold whenever a particular procedure is entered and a postcondition should hold whenever the procedure returns. An assertion can be placed inside a procedure body anywhere a statement can appear, and the asserted condition should hold whenever that point in the procedure is reached.

In Figure 2.2 (top) we have added a precondition, a postcondition and an assertion to our example procedure. The postcondition says that the procedure really does calculate the integer square root, that is, on termination we have $x^2 \leq N \leq (x+1)^2$.

```
procedure IntSqrt(N)

 @pre:   N >= 0;
 @post:  x*x <= N  &&  N < (x+1)*(x+1);

  {
    var x;

    x := 0;

    @assert x >= 0;

    while ((x+1)*(x+1) <= N)
      {
        x := x+1;
      }

    return x;
  }
```

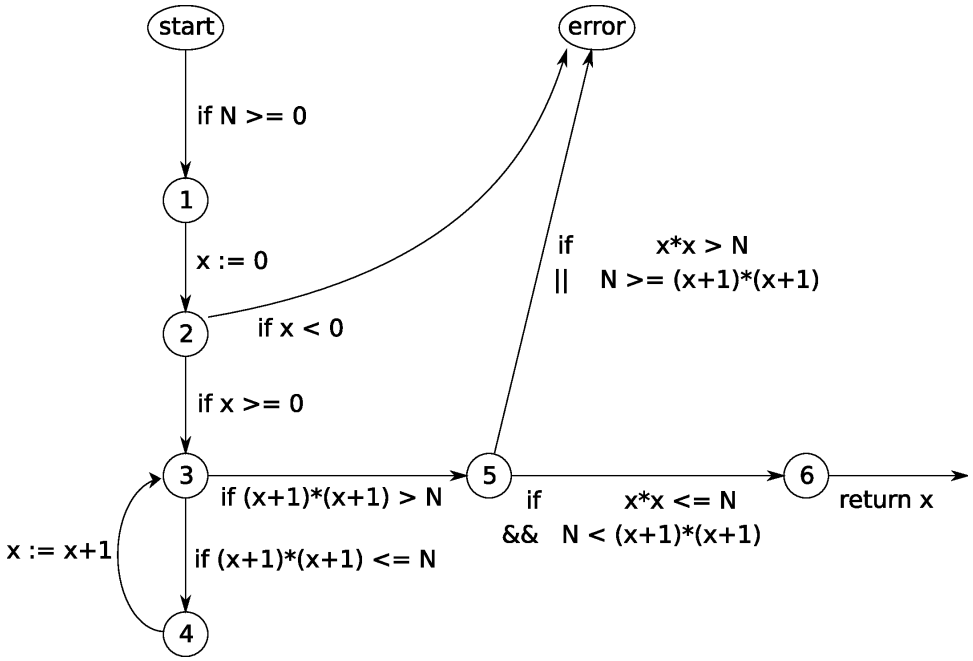

Figure 2.2: The procedure from Figure 2.1 annotated with a precondition, a post-condition and an assertion (top), and these annotations reflected in the control flow graph (bottom).

A standard approach to the problem of checking these specifications, used in e.g. [BR01], is to reduce it to the reachability problem, by appropriately transforming or *instrumenting* the program; this is what we will do.

Definition 2.1.1. The reachability problem is to determine, for a given set B of "bad" nodes in the CFG, whether execution can reach a node in B i.e. whether $reach(P)$ contains any elements of the form (l, s) where $l \in B$. □

Figure 2.2 (bottom) shows the CFG instrumented to reflect the specifications. We have added statements which check the asserted condition and postcondition: if the conditions are met, execution continues as before; if the conditions are violated execution is transferred to a special error node. Thus the error node is reachable in the instrumented CFG iff one of the specifications can be violated in the original program.

The instrumentation in Figure 2.2 (bottom) in effect allows the precondition to be *assumed* on entry rather than checked; alternatively, an extra transition to the error state can be used to check the precondition.

If the programming language contains instructions that may fail — e.g. in the case of dereferencing a null pointer, or performing an unsafe memory write — these failures may be treated in the same way, i.e. as transitions to a bad state. Thus we will automatically check programs for such low-level errors while we do assertion checking.

There are other ways to specify properties of programs, such as with temporal logic formulae which we will use later in Chapter 6 but we will concentrate on assertions. Finally we remark that the non-reachability of bad states is a *safety property* (e.g. [Sis94]): if a bad state is reachable, then there exists a finite execution trace which demonstrates this.

2.2 Decidability issues for verification

2.2.1 Finite state programs

Finite state programs are those for which the set of reachable states $reach(P)$ is finite. For such programs, the definition of $reach(P)$ gives an algorithm for computing it: starting with the initial states $\{start\} \times I$, repeatedly apply the function $X \mapsto X \cup semantics(P)(X)$, a process which is guaranteed to stabilise in a finite number of steps. Thus, the reachability problem for finite state programs is decidable: construct $reach(P)$, and check each element $(l, s) \in reach(P)$ in turn to see whether l is a bad node. If a bad node is reachable we can extract an execution trace leading to it with only a little more work.

Model checkers such as NuSMV [CCG+02] provide this functionality. The programming language accepted by NuSMV contains only finite data types and constructs, so we know *a priori* that all our programs are finite-state.

However, representing sets of (located) states explicitly, by enumerating their elements, and applying the program to one element after another, is very inefficient. Hence, NuSMV and other model checkers employ clever "symbolic" techniques, such as described in [BCM+92], using BDDs (binary decision diagrams) to represent both the program P and sets of states.

2.2.2 Beyond finite state: hello, undecidability

With infinite state programs — those where $reach(P)$ is infinite — we clearly cannot enumerate $reach(P)$ explicitly. Repeated application of $X \mapsto X \cup semantics(P)(X)$ will strictly increase forever. But if we represent sets of states symbolically, is it possible to compute (a reasonably explicit representation of) $reach(P)$?

For certain simple classes of infinite state programs, it is indeed possible. In [FS00]

for example, it is shown that for certain classes of two-counter automata, one can compute exactly the set of reachable states as a finite union of linear sets, a representation which is explicit enough to allow various properties to be checked.

Unfortunately, such results are very fragile. It is shown in [Min61] that as long as the programmer is allowed two natural number variables, and operations that add one, subtract one and test for zero, the reachability problem is undecidable. Note also that if a program location is reachable, then this is witnessed by a finite trace, of which (under very mild restrictions on the programming language) there are only countably many. Therefore any semi-decision procedure can be made into a decision procedure by running it in parallel with a systematic search [HC96, p.152]. Thus the "second prize" of a semi-decision procedure does not exist either.

To obtain decidable or semi-decidable checking of assertions, then, the restrictions on the programming language would need to be very severe; too severe to admit real-world applications programs.

2.3 Verification using inductive properties

Of course, showing that our verification problem is undecidable doesn't make it go away: society still needs correct software. So naturally research has focused on making verification systems which verify correct programs as often as possible. In this we are aided by the fact that the range of programs a (good) human programmer would ever write is a tiny subset of the theoretical range, consisting of the same basic idioms applied over and over [DDH72]. In particular, the programs used in practice generate states which are in some sense orderly, regular and full of patterns.

These efforts to make practically useful verification systems rest on two basic ideas: sound symbolic overapproximation and inductive properties. First we introduce the idea of inductive properties.

2.3.1 Induction with sets of states

Definition 2.3.1. A set $\Phi \subseteq S_{loc}$ of located states for program P is said to be *inductive* if the following conditions hold:

Initialisation: $(\{start\} \times I) \subseteq \Phi$

Consecution: $semantics(P)(\Phi) \subseteq \Phi$

Informally these conditions say that all initial states are in Φ, and that one cannot move from inside Φ to outside Φ by a single step of execution of P. □

The following fact explains our interest in inductive properties.

Remark 2.3.2. If Φ is inductive then

$$reach(P) \subseteq \Phi$$

i.e. Φ is a safe approximation, or overapproximation, of $reach(P)$. In particular, this means that if, for some set B of bad nodes,

$$\Phi \cap (B \times S) = \emptyset$$

then also

$$reach(P) \cap (B \times S) = \emptyset$$

i.e. the bad nodes are never reached. □

(We shall not prove this Remark or anything else in this chapter; in Chapter 3, when we develop in detail the theory of our verification system, we will properly prove instantiations/analogues of what is given here.)

The point of all this is that, if we can guess a suitable inductive property, we only need to check three conditions to verify the program, namely the two conditions

for inductiveness and $\Phi \cap (B \times S) = \emptyset$. This involves considering just one step of execution; no fixed points are needed.

In practice it is easier to decompose Φ into the sets of states located at each CFG node:

Remark 2.3.3. We can write Φ in a decomposed form such as

$$(\{start\} \times \Phi_{start}) \cup (\{1\} \times \Phi_1) \cup (\{2\} \times \Phi_2) \cup \ldots$$

where each Φ_l is a subset of S. The conditions for inductiveness then become:

Initialisation: $I \subseteq \Phi_{start}$

Consecution: For each edge from node n to node m, where n is labelled with statement i, we have $f(i)(\Phi_n) \subseteq \Phi_m$

and checking that $\Phi \cap (B \times S) = \emptyset$ amounts to checking that for each $b \in B$ we have $\Phi_b = \emptyset$. □

Figure 2.3 shows an inductive set Φ of states for our example program, decomposed in this way and placed alongside the CFG. Note that this Φ is an overapproximation of $reach(P)$ — for example Φ includes the state $(3, (N = 4, x = -2))$ which is not in $reach(P)$. Intuitively, we can see that the information about x being nonnegative is "thrown away" and not carried around the loop — but it is not needed: the inductive set given is small enough to verify the program, since it contains no elements of the form $(error, s)$.

2.3.2 General formulation of abstract induction-based verification

In the previous section we saw how programs can be verified with an appropriate inductive set of states. However, in that section we were still dealing explicitly with

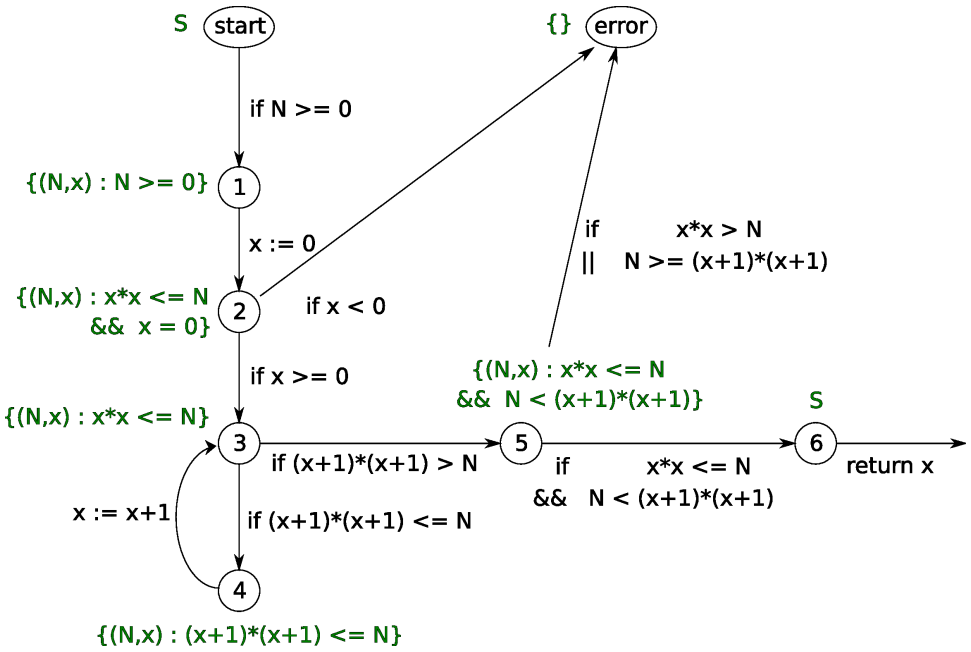

Figure 2.3: The instrumented CFG from Figure 2.2 along with an inductive set of states in green.

infinite sets of states, and we cannot compute in this way.

In this section we remedy that problem, introducing a general framework for abstraction, where infinite sets of states are represented and reasoned about symbolically.

Definition 2.3.4. An *abstraction domain* **D** consists of

1. a countable set A of abstract values
2. a concretisation function $\gamma : A \to \mathbb{P}(S)$
3. a distinguished element $\iota \in A$ such that $I \subseteq \gamma(\iota)$
4. a preorder (reflexive, transitive relation) $\preccurlyeq$ induced by γ, defined by

$$a_1 \preccurlyeq a_2 \iff \gamma(a_1) \subseteq \gamma(a_2)$$

5. a distinguished element $\bot \in A$ such that $\gamma(\bot) = \emptyset$
6. a computable operator $\sqcup : A \times A \to A$ such that $a_1 \preccurlyeq (a_1 \sqcup a_2)$ and $a_2 \preccurlyeq (a_1 \sqcup a_2)$

7. a computable binary relation $\sqsubseteq$ on A such that if $a_1 \sqsubseteq a_2$ then $a_1 \preccurlyeq a_2$

8. for each atomic program statement i with transfer function $f(i)$, an associated computable abstract transfer function $f(i)^\# : A \to A$ such that if $s \in \gamma(a)$ then $f(i)(s) \subseteq \gamma(f(i)^\#(a))$

The concretisation function γ gives meaning to the abstract values: $\gamma(a)$ is the set of states represented symbolically by a. The induced preorder $\preccurlyeq$ can be seen as the entailment relation between abstract values. Alternatively, it is common to speak of a relation like $\preccurlyeq$ as an *information order*: intuitively, the smaller the concretisation (image under γ) of an abstract value $a \in A$, the more precise a is, or the more information it expresses.

The operations $\sqcup$ and $f(i)^\#$ on A serve as sound approximations for $\cup$ and $f(i)$ on $\mathbb{P}(S)$. The element $\iota \in A$ approximates the initial states of the program. The relation $\sqsubseteq$ is a safe computable approximation of $\preccurlyeq$. If possible, we would like $\sqsubseteq$ to coincide with $\preccurlyeq$, but if the latter relation is uncomputable or very complicated we can make do with any safe stand-in. The element $\bot$ is used to label unreachable nodes. □

When working with multiple domains, we prevent confusion by referring to $\mathbf{D}$'s components as $\mathbf{D}.\gamma$, $\mathbf{D}.f(i)^\#$ and so on.

With this definition, we introduce the smallest amount of "machinery" that will allow us to tell the story of our background chapter; this simplifies the presentation. Similar but slightly more involved definitions are typically used in the literature: for the interested reader, [NNH99] gives several definitions similar to ours (see Sections 2.3, 6.1 and 6.3 of that work), including one as in [CC77], and examines their variations. One common additional assumption is that for any set $X \subseteq S$ of states, A contains a unique "best" abstract value to represent X. Formally, this means there exists $a \in A$ such that $X \subseteq \gamma(a)$ and for all $a' \in A$, if $X \subseteq \gamma(a')$ then $a \preccurlyeq a'$. Then an "abstraction function" $\alpha : \mathbb{P}(S) \to A$ is used to map each subset

$X \subseteq S$ to its best abstract representation. Another common assumption is that the abstract transfer functions are monotone (given a more precise approximation as input, they produce a more precise approximation as output). In return for such assumptions, one obtains additional guarantees about the analysis. In any case, when we introduce our verification framework in Chapter 3 we won't need to worry particularly about these issues, because our framework will treat all domains as though they are power sets, which are simple and familiar structures.

We now shadow the development of the previous Subsection, 2.3.1, but using abstract or symbolic values in place of explicit sets of states. The following Remark is the abstract parallel of Remark 2.3.3.

Remark 2.3.5. By assigning an abstract value (i.e. an element of A) at every CFG node, we abstractly represent a set of located program states. That is, we use a list

$$[(\mathit{start}, a_{\mathit{start}}), (1, a_1), (2, a_2), \ldots]$$

to represent the located state set

$$(\{\mathit{start}\} \times \gamma(a_{\mathit{start}})) \cup (\{1\} \times \gamma(a_1)) \cup (\{2\} \times \gamma(a_2)) \cup \ldots$$

The following two conditions are sufficient (but not necessary) for the set so represented to be inductive:

Initialisation: $\iota \sqsubseteq a_{\mathit{start}}$

Consecution: For each edge from node n to node m, where n is labelled with statement i, we have: $f(i)^{\#}(a_n) \sqsubseteq a_m$

It follows from inductiveness that if a CFG node is labelled with $\bot$ then execution of the program never reaches that node; if all "bad" nodes are labelled with $\bot$ then the program is correct. □

The important point is that the above conditions are computable, whereas those in Remark 2.3.3 are not.

Note that if a bad node is labelled with something other than $\bot$, then the preceding Remark 2.3.5 does not reveal anything about the program's correctness [CGL92]: perhaps the program is actually incorrect, but perhaps we just need to choose a more precise inductive set and then verification will succeed. Thus Remark 2.3.5 can "only" show that programs are correct, which is our main aim. If we want to show that programs are *incorrect* without generating an explicit counterexample, some other mechanism is needed; we will introduce one in Chapter 6.

2.3.3 Examples of abstract induction-based verification

Example 1: First-order formulae

When proving programs correct using the well-known Hoare logic [Hoa69, Apt81], and when using Dijkstra's calculus for deriving programs that are correct by construction [Dij75], first-order formulae serve as the symbolic representations of sets of states. We now show how the set of such formulae together with a sound theorem prover gives rise to an abstraction domain.

We take A to be the set of first order formulae, over some vocabulary appropriate for describing program states. The distinguished elements ι and $\bot$ are the formulae *true* and *false* respectively.

The concretisation function γ is given by the semantics of the logic:

$$\gamma(\Phi) := [\![\Phi]\!]$$

where $[\![\Phi]\!]$ is the set of states in which the formula Φ holds. Consequently the induced information order $\preccurlyeq$ is just entailment in the logic. The operator $\sqcup$ is simply the logical connective $\vee$.

It remains to say how to implement the relation $\sqsubseteq$ and the abstract transfer functions $f(i)^\#$. To implement $f(i)^\#$ we need to introduce the *strongest postcondition* operator as in [Bac88]. The strongest postcondition $SP(i)(\Phi)$ tells us everything we can know about the state which results from executing statement i in a state satisfying Φ. It is defined by the rules

$$\begin{aligned} SP(x := E)(P) &\triangleq \exists x'.P[x\backslash x'] \wedge x = (E[x\backslash x']) \\ SP(\text{if } Q)(P) &\triangleq P \wedge Q \end{aligned}$$

where $P[x\backslash E]$ denotes the substitution of every free occurrence of variable x in P by the expression E. (Throughout, we use $\triangleq$ for meta-equality when we wish to distinguish this from the equality predicate inside a logic, which is always written with the $=$ symbol.) We use SP to implement the abstract transformers:

$$f(i)^\#(\Phi) \triangleq SP(i)(\Phi)$$

We can implement $\sqsubseteq$ by using any sound theorem prover[1]: if the theorem prover can prove $\Phi_1 \rightarrow \Phi_2$ then we say $\Phi_1 \sqsubseteq \Phi_2$.

Figure 2.4 (top) demonstrates this abstraction domain on the integer square root program: we have assigned a formula (i.e. an abstract value) to each CFG node. Checking the abstract initialisation and consecution conditions from page 43, we see that (assuming the theorem prover can prove all the simple validities that arise) the located state set represented is inductive. Here we shall show just the consecution check for the edge from 4 to 3. Recall that we have to show that

$$f(x := x + 1)^\#(a_4) \quad \sqsubseteq \quad a_3$$

[1]Our use of "sound theorem prover" is intended to be as inclusive as possible; we also mean to include those programs described as decision procedures, SMT (*satisfiability modulo theories*) solvers etc..

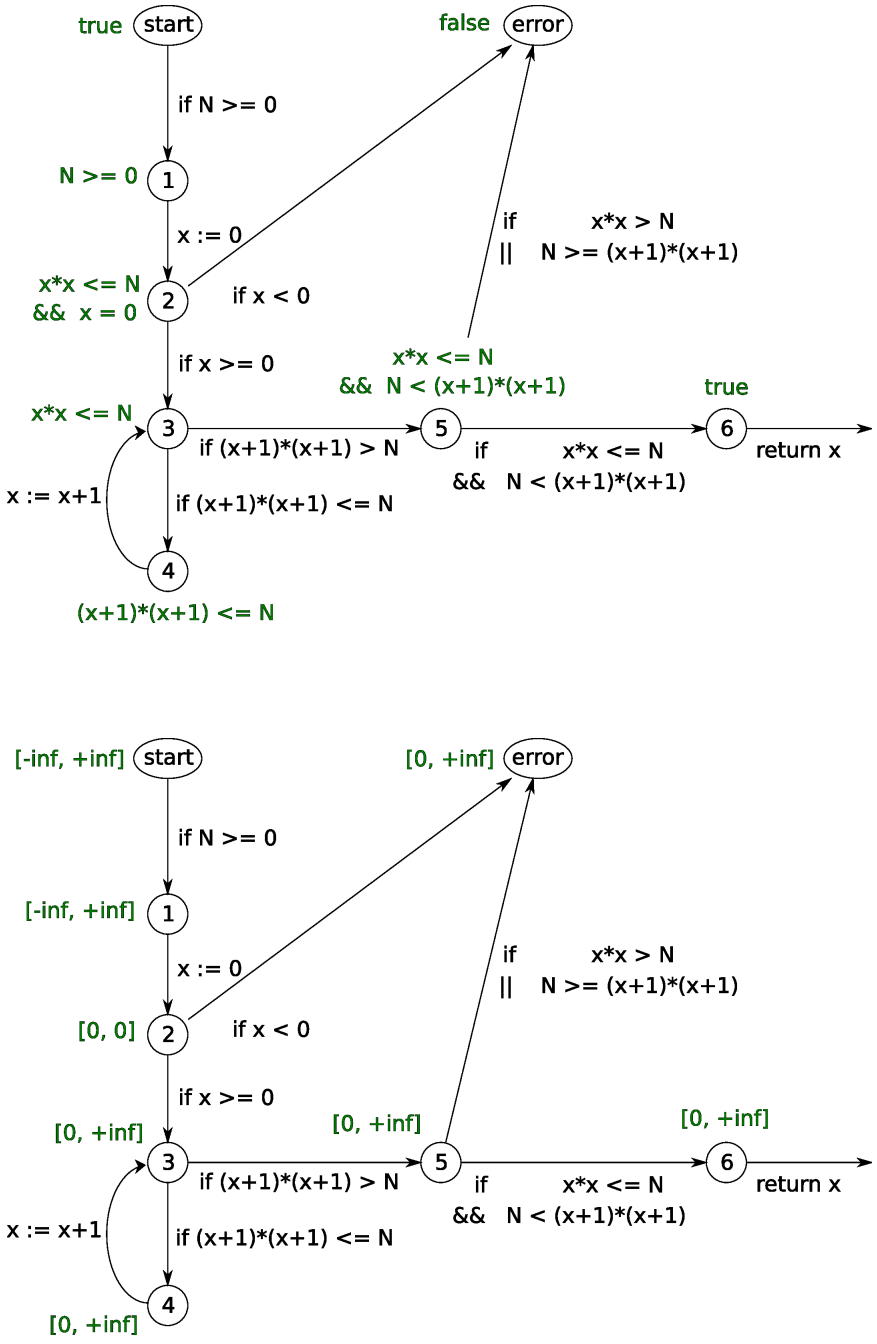

Figure 2.4: The integer square root CFG from Figure 2.2, with assignments of abstract values to the program's locations, using first-order formulae (top) and intervals (bottom) as the abstract values.

which expands as follows (using $\rightsquigarrow$ to mean "rewrites to" or "evaluates to")

$$\begin{array}{lll} \rightsquigarrow & f(x := x+1)^{\#}((x+1)^2 \leq N) & \rightarrow \quad x^2 \leq N \\ \rightsquigarrow & SP(x := x+1)((x+1)^2 \leq N) & \rightarrow \quad x^2 \leq N \\ \rightsquigarrow & \exists x'.(((x+1)^2 \leq N)[x \backslash x'] \wedge x = (x+1)[x \backslash x']) & \rightarrow \quad x^2 \leq N \\ \rightsquigarrow & \exists x'.((x'+1)^2 \leq N \wedge x = x'+1) & \rightarrow \quad x^2 \leq N \end{array}$$

and which a reasonable theorem prover could prove by noticing that x' must be $x-1$ and simplifying:

$$\begin{array}{lll} \rightsquigarrow & (x-1+1)^2 \leq N \wedge x = x-1+1 & \rightarrow \quad x^2 \leq N \\ \rightsquigarrow & x^2 \leq N \wedge x = x & \rightarrow \quad x^2 \leq N \\ \rightsquigarrow & x^2 \leq N & \rightarrow \quad x^2 \leq N \end{array}$$

Thus we have an inductive property. This, together with the fact that the bad node *error* is labelled with $\bot$ i.e. *false*, means we have verified the program.

(It is true that Figures 2.3 and 2.4 (top) *look* quite similar, but there is an important difference: in the first, nodes are explicitly labelled with infinite sets of states with which we cannot compute, whereas in the second the labels are formulae i.e. symbolic representations of sets of states, with which we can compute.)

In fact, in principle any reasonable logic over program states can be used in this way, though in practice a restricted first order logic is typically used due to the need for an effective automated theorem prover. With other logics it may also be necessary to use a sound approximation of the strongest postcondition, if this either cannot be expressed in the logic or cannot be obtained by a computable function.

Example 2: Intervals

We shall now apply a simple interval domain to our running example program, analysing only the variable x. Here we take our abstract space A to be

$$A \quad := \quad \left\{ [a,b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b \right\} \cup \{\bot\}$$

We interpret $[a,b]$ as an interval in $\mathbb{Z}$ which contains its end-points, i.e.

$$\gamma([a,b]) \quad := \quad \{(N,x) \mid a \leq x \leq b\}$$

and interpret the bottom element $\bot$, as usual, by $\gamma(\bot) = \emptyset$. Because the constraints are so simple, we can implement precise entailment checking, i.e. we can achieve $\sqsubseteq = \preccurlyeq$:

$$[a,b] \sqsubseteq [c,d] \quad \text{iff} \quad c \leq a \text{ and } b \leq d$$

Applying $\sqcup$ to two intervals generates the smallest interval that contains their union. The transfer functions perform interval arithmetic to track linear updates to the variables; we omit the details.

In Figure 2.4 (bottom) we show one inductive abstract property: all nodes in the loop have been assigned the interval $[0, +\infty]$. Intuitively, the only information carried around the loop is that x is nonnegative; everything else is "thrown away". This assignment of abstract values isn't precise enough to verify the postcondition and consequently at the bad state we don't have $\bot$. But the assertion that $x \geq 0$ can be verified in this way.

2.4 Obtaining inductive properties

2.4.1 Where do inductive properties come from?

In the previous section, we assumed that an assignment of abstract values to CFG nodes was somehow obtained, and discussed how to check that this assignment was inductive and verified the program. But where does such an assignment come from?

One approach to finding a suitable assignment is simply to ask the user to suggest it. In machine-checked Hoare logic proofs, for instance, the user must give a loop invariant for each loop. But providing these annotations imposes a heavy burden on programmers, which they do not seem prepared to embrace: whereas programmers are happy to annotate their variables with types in order to catch errors, asking for loop invariants appears too demanding. In any case, as computer scientists we are in the business of turning the work over to the computer.

Various methods for guessing likely invariants, which can then be checked, have been tried, including

simple heuristics: Here simple rules are used to generate likely invariants from the program; [Kal90] gives rules such as:

Replacing constants by variables: If the required postcondition refers to a constant (or a variable which is not modified in the loop body), obtain a candidate invariant by replacing it with a loop-modified variable

The Houdini program [FL01] implements such heuristics to propose invariants for Java programs.

dynamic invariant generation: Here we first record execution traces of the program running on a test suite. Then a set of relations between variables is generated, from a simple grammar, and checked against the stored execution

```
var worklist : node set
var a_start, a_error, a_1, ..., a_k : A

worklist := {start}
a_start := ι
for each v ∈ {error, 1, ..., k} do a_v := ⊥

while worklist ≠ ∅ do
    choose v ∈ worklist
    worklist := worklist − {v}
    for each edge from v to v' labelled with i do
        let a := f(i)#(a_v) in
            if not a ⊑ a_v' then
                a_v' := a_v' ⊔ a
                worklist := worklist ∪ {v'}
            end if
        end let
    end for
end while
```

Figure 2.5: Iterative worklist algorithm for forward propagation. This algorithm uses an arbitrary abstraction domain as defined in Definition 2.3.4.

traces. Any relations not refuted in one of the traces will be proposed as invariants. The most widely known implementation of this technique is the tool Daikon [PE04].

In this thesis we will use a technique called *forward propagation* to obtain inductive properties. Given an abstraction domain, this widely used technique finds strong (precise) assignments of abstract values that are inductive by construction.

2.4.2 Forward propagation

The idea of forward propagation [CC77][2] is to start with a very strong property (initially, assigning $\bot$ at all nodes other than *start*) and gradually weaken it until it becomes inductive. Figure 2.5 gives the algorithm. If this process terminates then we know that we have an inductive property.

[2]Forward propagation appears in [CC77], though the *term* "forward propagation" is not used for it there; this term appears in e.g. [SSM05, GT07, RCK07].

Because we start with a strong property and gradually weaken it, we hope to reach a strong solution, and therefore have more chance of verifying the program. Indeed, under certain conditions it can be shown that the above iterative process produces, after ω iterations, the best solution (e.g. [NNH99]). Unfortunately, of course, the process doesn't always terminate, as the following example shows. The next section gives a way to attack this non-termination problem.

Let's see what happens if we run the forward propagation algorithm on our example program, using interval analysis. Initially, we have $a_{start} = \iota = [-\infty, +\infty]$ and $\bot$ for all other nodes, and the worklist contains just node *start*. So we remove node *start* from the worklist, and generate its successor for the edge from *start* to 1 labelled with `if N >= 0`:

$$f(\text{ if } N \geq 0)^{\#}([-\infty, +\infty]) = [-\infty, +\infty]$$

We then see whether this new value, $[-\infty, +\infty]$, is "covered" by the existing value at 1 using $\sqsubseteq$. We don't have $[-\infty, +\infty] \sqsubseteq \bot = a_1$, and so we need to merge the two values with $\sqcup$. This gives $[-\infty, +\infty] \sqcup \bot = [-\infty, +\infty]$ as the new value of a_1, and causes node 1 to be added to the worklist.

Next we remove node 1 from the worklist, and generate the successors for the edge from 1 to 2 labelled with `x := 0`:

$$f(x := 0)^{\#}([-\infty, +\infty]) = [0, 0]$$

This isn't covered by the value currently at node 2, as we don't have $[0, 0] \sqsubseteq \bot = a_2$, so we set the new value at 2 to $[0, 0] \sqcup \bot = [0, 0]$ and add node 2 to the worklist. At this point we have reached the state shown in Figure 2.6(a).

We will now see the reason for non-termination. After the algorithm has propagated around the loop once, we reach the situation in Figure 2.6(b). From there, we generate $f(\text{ if } (x + 1)^2 \leq N)^{\#}([0, 1]) = [0, 1]$ so a_4 becomes $[0, 1] \sqcup [0, 0] = [0, 1]$. Then we generate $f(x := x + 1)^{\#}([0, 1]) = [1, 2]$ so a_3 becomes $[1, 2] \sqcup [0, 1] = [0, 2]$.

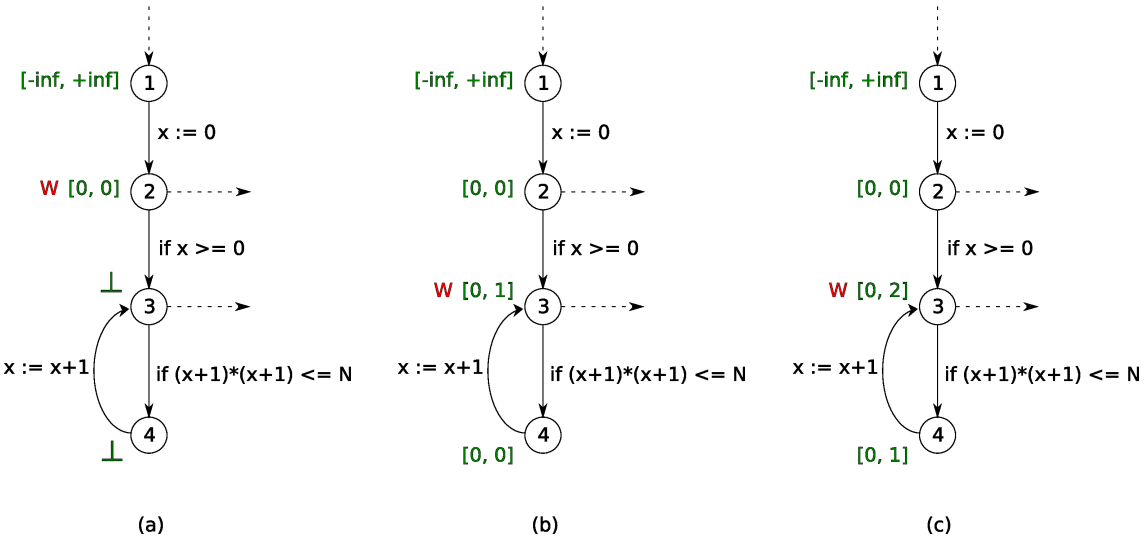

Figure 2.6: Part of the CFG of our example program (from Figure 2.2) shown at three stages in the running of the forward propagation algorithm (Figure 2.5) using an interval analysis for x. Nodes marked with W are in the worklist.

At this point we reach the situation shown in Figure 2.6(c), which is the same as (b) except the top end of the intervals has increased by one. In this way the algorithm will move around the loop again and again, generating $[0, 3]$, then $[0, 4]$, and so on, and the process will not terminate.

2.4.3 Finite and finite-height domains ensure termination

In the example above, the computation failed to stabilise because at each iteration we only weakened the abstract constraints a very little bit, so the computation approached the limit "very slowly" and never reached it. To avoid this we need to "accelerate" the computation, so that "bigger steps" are taken on some iterations and the computation stabilises.

Much existing work uses *widening operators* [CC76, CC92b] for this purpose. A *widening operator* ∇ works like $\sqcup$ except that it takes bigger steps; under appropriate conditions the use of widening operators guarantees the convergence of the computation.

In this thesis, however, we shall stick to a simpler approach, which is to make the

abstraction domain *finite-height*. We say that an abstraction domain is finite-height if every $\preccurlyeq$-increasing sequence of abstract values is eventually constant. In particular this must be true if A is finite.

To make the interval domain finite-height, we might decide that whenever we see an interval with an end-point below -100 or above 100, we will weaken this to $-\infty$ or ∞ respectively, that is to say, we might replace A by the subset

$$A' \quad := \quad \{[a,b] \in A \mid a \geq -100 \text{ or } a = -\infty, \text{ and } b \leq 100 \text{ or } b = \infty\} \cup \{\bot\}$$

Using this finite-height domain, the forward propagation algorithm terminates on our example program, giving the inductive property shown (by the green values) in Figure 2.4 (bottom).

Finite-height abstractions are said to be less powerful than widening operators [CC92b]. On the other hand, they are conceptually simpler.

2.5 Survey of well-known abstraction domains

In the previous section we saw how, once we have chosen an abstraction domain and either made it finite-height or supplied a widening operator, the forward propagation algorithm can automatically discover inductive properties.

Successful use of forward propagation depends crucially on the choice of a "good" (i.e. appropriate) abstraction domain: if the abstraction retains too much irrelevant information this results in high computational cost, but if relevant facts are thrown away it will not be possible to verify the program. This is exactly the problem we concern ourselves with in this thesis: by allowing the user to build complex domains as *ad-hoc* combinations of existing components, we hope to make it easier to construct an appropriate domain.

We shall introduce the idea of domain combination shortly. But before that, in this section we survey some of the well-known abstraction domains presented in the literature (we cannot hope to be exhaustive), roughly divided into the following categories:

1. Predicate abstraction
2. Classical program analyses
3. Systems for shape analysis
4. Numerical domains

2.5.1 Predicate abstraction

Monomial predicate abstraction

Predicate abstraction is essentially a disciplined, finite-domain way to use a logic over program states. The idea is to group the program states into equivalence classes based on the truth values they give to a finite collection of predicates. We choose *abstraction predicates* $P^1, \ldots, P^n$ and then abstract each state s to the formula $\Psi \triangleq \Psi^1 \wedge \ldots \wedge \Psi^n$ where

$$\Psi^i \triangleq \begin{cases} P^i & \text{if } P^i \text{ is true in s} \\ \neg P^i & \text{if } P^i \text{ is false in s} \end{cases}$$

Such formulae Ψ are called *monomials*. Each monomial over n abstraction predicates can be succinctly represented as a vector of n bits, where the ith bit records the polarity of P^i, and sets of monomials can be succinctly represented using BDDs.

In *monomial predicate abstraction*, elements a of the set A are *sets of* monomials formed as above. These are sufficient to express any boolean combination of the

abstraction predicates. Because only 2^n monomials can be made from n abstraction predicates, this domain is finite. To concretise such a set of monomials, we just take the truth set of each monomial, and form their union.

$$\gamma(a) \quad \triangleq \quad \bigcup_{\Phi \in a} [\![\Phi]\!]$$

The element $\bot$ is the empty set and $\sqsubseteq$ is the subset relation $\subseteq$.

The transfer functions can be implemented using a theorem prover for the logic and the strongest postcondition operator, as the following example shows.

Example 2.5.1. Consider the statement `x := x - 1` in a program with a single integer variable, and suppose we have chosen two abstraction predicates $P^1 \triangleq x^2 = 1$ and $P^2 \triangleq x > 0$. Let us find the successors under that statement of the single monomial $P^1 \wedge P^2$, i.e. let us calculate

$$f(x := x - 1)^{\#}(\{P^1 \wedge P^2\})$$

Should we include $\neg P^1 \wedge P^2$ in the successor set of monomials? The implication

$$SP(x := x - 1)(P^1 \wedge P^2) \quad \rightarrow \quad \neg(\neg P^1 \wedge P^2)$$

states that after executing `x := x - 1` in a state satisfying $P^1 \wedge P^2$ we *cannot* be in a state satisfying $\neg P^1 \wedge P^2$. This expands as follows:

$$\begin{array}{lll}
\rightsquigarrow & \exists x'.(P^1 \wedge P^2)[x \backslash x'] \wedge x = (x-1)[x \backslash x'] & \rightarrow \quad \neg(\neg P^1 \wedge P^2) \\
\rightsquigarrow & \exists x'.(x^2 = 1 \wedge x > 0)[x \backslash x'] \wedge x = (x-1)[x \backslash x'] & \rightarrow \quad \neg(\neg x^2 = 1 \wedge x > 0) \\
\rightsquigarrow & \exists x'.(x'^2 = 1 \wedge x' > 0) \wedge x = x' - 1 & \rightarrow \quad \neg(\neg x^2 = 1 \wedge x > 0)
\end{array}$$

and from here a reasonable theorem prover could prove the formula by noting that x' must be $x + 1$ and simplifying down to a trivial implication:

$$\begin{array}{lll} \rightsquigarrow & (x+1)^2 = 1 \wedge x + 1 > 0 & \rightarrow \quad x^2 = 1 \vee x \leq 0 \\ \rightsquigarrow & (x+1 = 1 \vee x + 1 = -1) \wedge x + 1 > 0 & \rightarrow \quad x^2 = 1 \vee x \leq 0 \\ \rightsquigarrow & (x = 0 \vee x = -2) \wedge x \geq 0 & \rightarrow \quad x^2 = 1 \vee x \leq 0 \\ \rightsquigarrow & x = 0 & \rightarrow \quad x^2 = 1 \vee x \leq 0 \end{array}$$

Because this formula is proved, we should not include $\neg P^1 \wedge P^2$ in the successor set. On the other hand, we *should* include $\neg P^1 \wedge \neg P^2$: the corresponding formula is

$$\begin{array}{lll} & SP(x := x - 1)(P^1 \wedge P^2) & \rightarrow \quad \neg(\neg P^1 \wedge \neg P^2) \\ \rightsquigarrow & \exists x'.(P^1 \wedge P^2)[x \backslash x'] \wedge x = (x-1)[x \backslash x'] & \rightarrow \quad \neg(\neg P^1 \wedge \neg P^2) \\ \rightsquigarrow & \exists x'.(x^2 = 1 \wedge x > 0)[x \backslash x'] \wedge x = (x-1)[x \backslash x'] & \rightarrow \quad \neg(\neg x^2 = 1 \wedge \neg x > 0) \\ \rightsquigarrow & \exists x'.(x'^2 = 1 \wedge x' > 0) \wedge x = x' - 1 & \rightarrow \quad \neg(\neg x^2 = 1 \wedge \neg x > 0) \end{array}$$

which no sound theorem prover will prove because it is not valid (e.g. put $x' = 1$ and $x = 0$).

Note that this approach implicitly takes into account the possibility of an incomplete theorem prover: if the prover fails to prove an implication which is valid (perhaps the prover "times out" without discovering a proof), this results in an extra successor state being generated, which is less precise but still sound. Incompleteness of the prover can never cause fewer successors to be generated.

Predicate abstraction can discover a wide range of relationships between variables (compared to interval analysis for instance), and has been successful: for example, it is now used commercially by Microsoft to verify interface usage properties of device drivers, in a tool called Static Driver Verifier (SDV) [BBC+06]. Well-known implementations of predicate abstraction for C programs are SLAM [BR01] (which

is the "heart" of SDV), BLAST [HJMS02] and MAGIC [CCG+04].

However there are also problems with predicate abstraction. Firstly, predicate abstraction tends to be based on first-order logics, which cannot express many important heap properties needed to reason about programs that use linked data structures (see later Section 2.5.3). Secondly, there is the issue of how to obtain "good" choices for the abstraction predicates: are these suggested by the user, or is some automatic method tried?

Finally, the computational cost of predicate abstraction is relatively high because successor computations require many invocations of a theorem prover. In the worst case, each abstract state can contain exponentially many monomials (in the number n of abstraction predicates), and for each one of these, an exponential number of calls to the theorem prover is needed (one for each potential successor monomial).

Trivector predicate abstraction

One way to understand the need for an exponential number of theorem prover calls is as follows. Intuitively, each time monomial predicate abstraction cannot determine whether an abstraction predicate P^i will be true or false in the successor state, a "case split" is performed giving two lots of successor states: one with P^i (the undetermined predicate holds) and one with $\neg P^i$ (the undetermined predicate does not hold). With n abstraction predicates, up to n such case splits are possible, giving 2^n cases.

Trivector predicate abstraction [BPR01] is a variant of predicate abstraction in which an undetermined abstraction predicate no longer results in a case split: such a predicate is simply omitted from the successor constraint. This reduces the computation cost: now only one formula need be maintained at each CFG node (as opposed to a set of formulae) and successor computations need only a linear number of theorem prover calls. However, precision is lost, because relationships between the

abstraction predicates are no longer tracked.

In trivector predicate abstraction, abstract values $a \in A$ are formulae (not sets of formulae) of the form $\Psi^1 \wedge \ldots \wedge \Psi^n$ where each Ψ^i is one of P^i, $\neg P^i$, *true*. Each such formula can be succinctly represented as a vector of n trits (ternary digits), or trivector. Concretisation is trivial because each $a \in A$ is already a formula: $\gamma(a) = [\![a]\!]$. Here we give an example of how a theorem prover, together with the strongest postcondition operator, is used to calculate the successor functions.

Example 2.5.2. Consider as before the statement `x := x - 1` in a program with a single integer variable, and two abstraction predicates $P^1 \triangleq x^2 = 1$ and $P^2 \triangleq x > 0$. Let us find the successors under that statement of $\neg P^1 \wedge P^2$, i.e. let us calculate

$$f(x := x - 1)^{\#}(\neg P^1 \wedge P^2)$$

After the statement has executed we know

$$SP(x := x - 1)(\neg P^1 \wedge P^2)$$

which expands to

$$\begin{aligned} &\rightsquigarrow && SP(x := x - 1)(\neg x^2 = 1 \wedge x > 0) \\ &\rightsquigarrow && \exists x'.(\neg x^2 = 1 \wedge x > 0)[x \backslash x'] \wedge x = (x - 1)[x \backslash x'] \\ &\rightsquigarrow && \exists x'.\neg x'^2 = 1 \wedge x' > 0 \wedge x = x' - 1 \end{aligned}$$

Now we check each of the abstraction predicates in turn, to see whether it or its negation is entailed by the above. For P^1 we invoke the theorem prover on the implication

$$\big(\exists x'.\neg x'^2 = 1 \wedge x' > 0 \wedge x = x' - 1\big) \quad \rightarrow \quad x^2 = 1$$

which cannot be proved (as it is not valid; put e.g. $x' = 3$, $x = 2$) and then on

$$(\exists x'.\neg x'^2 = 1 \wedge x' > 0 \wedge x = x' - 1) \quad \rightarrow \quad \neg x^2 = 1$$

which also cannot be proved (as it is not valid; put e.g. $x' = 2$, $x = 1$). Thus P^1 will be omitted from the successor, that is, Ψ^1 will be *true*.

But P^2 *will* be included in the successor, as a reasonable theorem prover will prove

$$\exists x'.\neg x'^2 = 1 \wedge x' > 0 \wedge x = x' - 1 \qquad \rightarrow \quad x > 0$$

e.g. by noting that x' must be $x + 1$ and simplifying as follows:

$$\begin{array}{lll}
\rightsquigarrow & \neg(x+1)^2 = 1 \wedge x + 1 > 0 & \rightarrow \quad x > 0 \\
\rightsquigarrow & \neg(x+1 = 1 \vee x + 1 = -1) \wedge x \geq 0 & \rightarrow \quad x > 0 \\
\rightsquigarrow & \neg(x = 0 \vee x = -2) \wedge x \geq 0 & \rightarrow \quad x > 0 \\
\rightsquigarrow & x \neq 0 \wedge x \neq -2 \wedge x \geq 0 & \rightarrow \quad x > 0 \\
\rightsquigarrow & x \neq 0 \wedge x \geq 0 & \rightarrow \quad x > 0 \\
\rightsquigarrow & x > 0 & \rightarrow \quad x > 0
\end{array}$$

Thus we have

$$f(x := x - 1)^{\#}(\neg P^1 \wedge P^2) = P^2$$

compared to the result for monomial predicate abstraction, where there will be a "case split" on P^1:

$$f(x := x - 1)^{\#}(\{\neg P^1 \wedge P^2\}) = \{\neg P^1 \wedge P^2, P^1 \wedge P^2\}$$

Note that again any incompleteness in the theorem prover is implicitly handled safely.

2.5.2 Classical program analyses

Like software verification, the field of *program analysis* [NNH99] is concerned with extracting from a source program some semantic information describing its operation. The difference, broadly speaking, is that program analysis is concerned with detecting low-level properties which enable programs to be compiled to efficient code, whereas software verification is concerned with high-level properties meaningful to a system designer. Making a program run quickly is not the job of verification.

Still, some algorithms from program analysis may suit our purpose because, although they infer fairly shallow properties, they do so quickly, and these properties may aid the operation of a subsequent deeper analysis.

1. **Alias analysis:** *Aliasing* is when two syntactically distinct expressions (perhaps two pointer or reference variables) refer to the same memory location. Alias analysis gives a conservative approximation of which pairs of expressions might be aliased at each program point. Many algorithms are known for this problem: see for example [WWA+01].

2. **Constant propagation:** (e.g.[WZ91]) Here one tries to detect expressions that are constant at a particular program point, i.e. will always evaluate to the same value at that point.

3. **Mod/Ref analysis:** This analysis discovers which parts of a program's store (i.e. which variables and which heap locations) might be referenced or modified by each part of the program. A notable paper in this regard is [SR05], which shows how to conservatively characterise the objects that might be modified by a given Java method.

2.5.3 Linked data structures and shape

It has been part of programming folklore for a long time (e.g. [Hoa73]) that programs which use pointers and linked data structures are easy to get wrong, and difficult to reason about. The most immediate difficulty is that Hoare logic's elegant substitution axiom scheme for assignment

$$\vdash \{P[x\backslash E]\}\ x := E\ \{P\}$$

does not apply if we replace the variable x on the left hand side with a field access, e.g. as in $x.f := E$. This is because of *aliasing*: if another variable y is equal to x (is an alias for x) then the value of $y.f$ will also change, yet this is not reflected in the substitution rule. Morris' general assignment axiom scheme [Mor82], which essentially substitutes on the basis of semantic rather than syntactic identity, allows reasoning about assignments made via pointers: defining $P_E^{/x}$ to be the result of replacing each reference y in P with

$$\text{if } \mathit{address}(y) = \mathit{address}(x) \text{ then } E \text{ else } y$$

we have

$$\vdash \{P_E^{/x}\}\ x := E\ \{P\}$$

Yet reasoning about pointer programs is still very difficult. Each time we need to use the above assignment rule, we need to derive a set of equalities and inequalities between pointer variables. Where do these come from? Also, in general we are interested in verifying specifications that describe the *structure* of the heap. For example, the postcondition of a routine that inserts a new element into a linked list will say that the new element is now reachable from the list head by following some number of 'next' pointers. From [Mor82]:

> "The formal treatment of linked data structures is significantly more

> complex than that of simpler structures because it is necessary to handle not just properties of individual nodes, but also relationships between the nodes.
>
> The most fundamental of these relationships is that of connectivity. If the verification of programs manipulating linked data structures is to be manageable, then it is essential that connectivity relations can be comfortably maneuvered through changes in the data structure."

What is needed is some way to represent the configuration, or topology, or "shape" of the heap. This includes, in particular, which nodes can be reached from which others. We now survey some of the abstractions proposed in the literature for representing the shape of the heap.

Transitive closure logic

Suppose we need to express a reachability condition, such as that the (object pointed to by) variable v can be reached from the (object pointed to by) variable u by following f-fields. First order logic with transitive closure [Imm98], or FO(TC), augments first order logic with an operator for expressing such properties: formulae of the form

$$TC_{[a,b]}\left[\Phi(a,b)\right](t,t')$$

are true just when there is some finite sequence of points starting at t and ending at t', such that each pair of successive points (a,b) in the sequence satisfies $\Phi(a,b)$. FO(TC) is desirable because it is very expressive, e.g. we can write

- x points to an acyclic list (using f-edges):
  $TC_{[a,b]}\left[f(a)=b\right](x,null)$
- Only the g-fields of objects transitively reachable from x by f-fields may be null:

T_1 axiom scheme:

$$\forall u, v: \ TC_{[a,b]}\left[\Phi(a,b)\right](u,v) \ \leftrightarrow \ (u=v) \vee \big(\exists w: \ \Phi(u,w) \wedge TC_{[a,b]}\left[\Phi(a,b)\right](w,v)\big)$$

Induction axiom scheme:

$$\begin{array}{l}(\forall z: \ \ Z(z) \rightarrow P(z)) \ \wedge \ (\forall u, v: \ \ P(u) \wedge \Phi(u,v) \rightarrow P(v)) \\ \rightarrow \ \forall u, z: \ \ Z(z) \wedge TC_{[a,b]}\left[\Phi(a,b)\right](z,u) \rightarrow P(u)\end{array}$$

"No exit" colouring axiom scheme:

$$\begin{array}{c}\forall u, v: \ \ A(u) \wedge \neg A(v) \ \ \rightarrow \ \ \neg\Phi(u,v) \\ \rightarrow \ \ \forall u, v: \ \ A(u) \wedge \neg A(v) \ \ \rightarrow \ \ \neg\, TC_{[a,b]}\left[\Phi(a,b)\right](u,v)\end{array}$$

Figure 2.7: Some sound but incomplete first order axioms for transitive closure, taken from [LAIR+05]

$$\forall o \ \ g(o) = null \ \ \rightarrow \ \ TC_{[a,b]}\left[f(a) = b\right](x,o)$$

To reason about such properties, one could use induction on natural numbers, but this is not done in practice because automating induction proofs is difficult and such proofs would be required in each small step of the analysis. Instead, a common idea is to use induction to manually derive some sound (but incomplete) rules about reachability, such as those in Figure 2.5.3. These are then given to an ordinary first order theorem prover, which is freed from considering induction or the natural numbers at all. This approach is taken in [Mor82, Nel83] and recently in [LAIR+05, LQ06].

Decidable logics with reachability

In addition to searching for heuristic techniques for reasoning about reachability, one can investigate whether, by restricting the logic and/or the class of models, one may find a logic with reachability which is decidable and yet sufficiently expressive to be worthwhile. We can then simply throw all queries in such a logic to a decision procedure. The ability to express forms of reachability appears in logics in various guises: explicitly as in the F operator of LTL [Pnu77], as fixed-point operators, as

inductively defined predicates, as transitive closure operators and in second-order logics which can quantify over sets.

Unfortunately, even gentle logics tend to become undecidable when reachability is added. For example, first order logic with two variables is decidable, but adding transitive closure leads to undecidability [GOR97]. Nevertheless, several decidable logics which can express some form of reachability are known, with various strengths and weaknesses, including *WS2S* [KM01], $\exists\forall(DTC^+[E])$ [IRR+04], the guarded fixed point logic μGF [GW99] (which includes the modal μ-calculus [Koz83]), and LRP_2 [YRS+06].

Logics with some form of reachability can be used with the predicate abstraction technique; see [DN03] for an example of this.

PALE and graph types

The Pointer Assertion Logic Engine (PALE) [MS01] tool is used to verify that procedures manipulating graph type data structures preserve their consistency. A *graph type* data structure [MS01] consists of some acyclic tree *backbones* augmented by some well-behaved "extra" pointers governed by a datatype invariant. A linked list where each node has a pointer to the last node is a graph type, as is a binary tree where the leaves are threaded into a cyclic list.

Graph type heaps can be encoded conveniently as models of the tree logic *WS2S* because the tree structure needed to handle the backbones is already built in. PALE accepts programs in a C-like language, ignoring arithmetic statements. The programmer must provide loop invariants and a special graph type declaration for each type used.

Figure 2.8 contains a graph type declaration for doubly linked lists. The 'next' fields

```
type Node = {
  bool value;
  data next:Node;
  pointer prev:Node[this^Node.next={prev}];
}
```

Figure 2.8: An example of a graph type declaration for the PALE tool. Here we declare nodes to contain a Boolean data field `value` and pointer fields 'next' and 'prev', which we constrain to be inverses of each other with the declaration `pointer prev:Node[this^Node.next={prev}]`. (Here `^Node.next` indicates a backward step along the 'next' field.)

form the backbone, and the 'prev' fields are extra pointers; the declaration

```
pointer prev:Node[this^Node.next={prev}]
```

constrains these to be the inverses of the 'next' fields (`^` is the "backwards" operator, so `this^Node.next` denotes starting at `this` and going one step backwards along field `Node.next`). PALE generates verification conditions in *WS2S* and sends them to the MONA tool [KM01] which decides them using tree automata. Thus — within its limited domain of application — the shape analysis of PALE is utterly precise.

TVLA and three-valued shape graphs

In this approach, taken by the TVLA (Three-Valued Logic Analyser) system [SRW99, LAMS04], sets of concrete heaps are represented by three-valued models.

With each concrete program state we associate a model of a predicate logic with unary and binary predicates, as in Figure 2.9. The universe of these models represents the set of allocated objects. For each (pointer-typed) program variable v there is a unary predicate V which holds only at the object pointed to by v. Similarly, pointer-typed fields are represented by binary predicates. We may choose to have additional predicates, such as unary predicates giving the types of objects. Note that the data fields of objects are abstracted away.

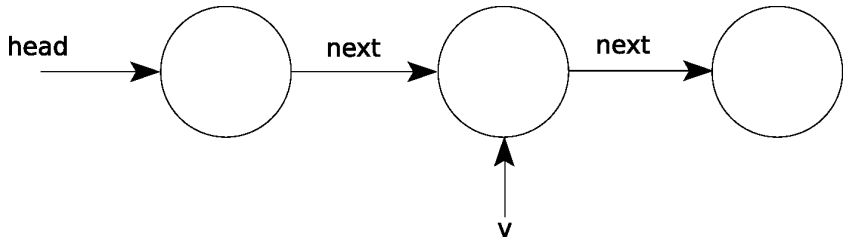

Figure 2.9: A concrete heap in TVLA

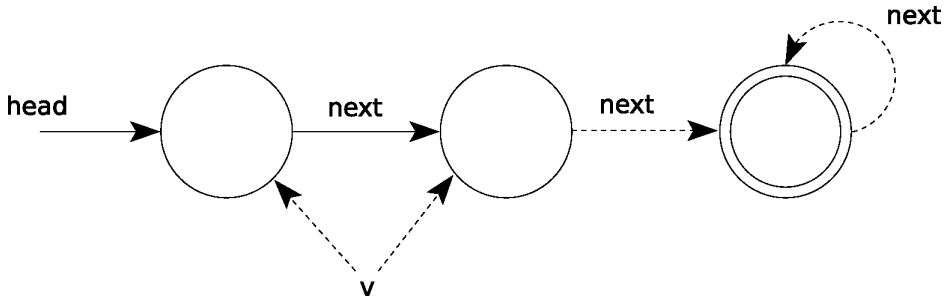

Figure 2.10: An abstract heap, representing a linked list of length three or more, with v pointing to the first or second element. This is one abstraction of the concrete heap in Figure 2.9

To give the semantics for an instruction i we provide an *update rule* for each predicate changed by i. These express the values of the predicates after execution of i in terms of their values beforehand and may make use of transitive closure. For instance the instruction $v := u.f$ has the update rule

$$V(o) := \exists p(U(p) \wedge F(p, o))$$

Abstraction is achieved by moving to a three-valued semantics, where there is an extra truth value *unknown* in addition to the usual *true* and *false*. Appendix A.1 gives an introduction to three-valued logic for readers unfamiliar with it.

In abstract states, such as the one in Figure 2.10, predicates may take the value *unknown*, which is depicted as a dashed line. Dashed edges show where a variable or field *might* point, or *is allowed to* point, without *requiring* this to be the case. Nodes representing one or more concrete nodes, called *summary nodes*, are also allowed, and are drawn with a double circle. The abstract heap in Figure 2.10 represents all linked lists of length three or more starting at *head* and where v points to the first

or second element; this includes the heap in Figure 2.9.

Sound abstract transfer functions are obtained automatically simply by interpreting the update rules in three-valued logic. In addition to the *core* predicates, which are used to define the semantics of the language, one may introduce additional *instrumentation* predicates, and associated update rules, to increase the precision of abstract heaps. Commonly these record reachability properties.

Separation logic

Separation logic [ORY01] addresses the difficulties in using Morris' axiom in a different way. Its characteristic feature is that it extends first order logic with a new connective, $\star$, the *separating* or *spatial conjunction*. The formula $P \star Q$ is satisfied by heaps which can be partitioned into a part satisfying P and a part satisfying Q. For example, using the predicate $\mathit{Tree}(v)$ to denote a binary tree rooted at v, the formula

$$\mathit{Tree}(x) \star \mathit{Tree}(y)$$

means that the heap contains two disjoint trees rooted at x and y respectively. Since the two trees must be disjoint, this implies that $x \neq y$.

The key point is that once we have partitioned the heap in this way, we can make a change to one of the sections and we know that properties of the other sections will be preserved. This idea is embodied in the *frame rule*:

$$\frac{\{P\}C\{Q\}}{\{P \star R\}C\{Q \star R\}} \mathit{modifies}(C) \cap \mathit{free}(R) = \emptyset$$

where $\mathit{free}(R)$ denotes the free variables in the formula R, and $\mathit{modifies}(C)$ denotes the variables modified by the program fragment C. For example, suppose $\mathit{dispose}(v)$ is a routine that deallocates a binary tree rooted at v, specified by

$$\{\mathit{Tree}(v)\}\ \mathit{dispose}(v)\ \{\mathit{empty}\}$$

(where *empty* is the empty heap, which is a unit of $\star$). Then the frame rule allows us to infer

$$\{\mathit{Tree}(x) \star \mathit{Tree}(y)\}\ \mathit{dispose}(y)\ \{\mathit{Tree}(x) \star \mathit{empty}\}$$

The nice thing about separation logic is that weakest preconditions for heap update can be expressed very neatly, so a growing number of manual proofs have been done this way, such as in [BTSR04, PBO07]. Some decidable fragments of separation logic are known [BCO05b], and separation logic has been used for automated program verification in the Smallfoot tool [BCO05a].

Other domains for the heap

Numerous other domains for reasoning about the shape of the heap have been proposed, including for example

ownership types: The concept of *ownership* (e.g. [Wre03]) can help structure the heaps of object-oriented programs. Suppose O and P are objects at the top level of a program. Informally, for O to *own* P means that O controls access to P. For example, a Map object may use a binary tree representation, and will own the tree nodes it uses internally; objects from "outside" have no business holding pointers to these tree nodes, as they should only affect the data structure by invoking methods of the Map object. Universe types [DM05, CDE07, CDD+08], among others, have been developed for checking ownership properties (mostly) statically.

regular model checking: Here program heaps are represented as finite words over a finite alphabet [BHMV05]. The abstract values a are then finite state automata (equivalently, regular expressions) over this alphabet; the interpretation $\gamma(a)$ of such an automaton is the set of heaps whose encodings it accepts.

To obtain the abstract transfer function $f(i)^\#$ for an instruction i, we construct

a finite state transducer which captures the effect of i (as a regular relation). This transducer can then be composed with automata.

graph grammars: One can generalise from regular languages to more complicated ones. [FM97], for example, uses a kind of context free graph grammar to represent sets of heaps.

buffer domain: these provide a specialised representation to track the use of string buffers in C (contiguous areas of memory, often null-terminated) [SK02].

2.5.4 Numerical domains

A variety of abstract domains are known for discovering linear relationships between numerical variables $x_1, \ldots, x_n$ in a program:

- **Interval analysis:** [CC76, SW04] Here we associate an interval $[a_i, b_i]$ with each x_i. We saw this on page 48 (albeit for a single variable). Thus, each constraint has the form

$$\bigwedge_{i=1}^{n} a_i \leq x_i \leq b_i$$

  Interval analysis is fast and simple, and each constraint needs only $O(n)$ space; on the other hand, it does not discover dependencies between the different variables.

- **Polyhedral analysis:** [CH78, BHRZ03] Here each constraint is a convex polyhedron containing the values of the variables, i.e. it is a conjunction of "half-spaces" each having the form $c_1x_1 + \ldots + c_nx_n \leq a$.

  As in the interval case, this class of constraints is not closed under union, so $\sqcup$ produces the convex hull of its operands. Polyhedral analysis is much more precise, capturing complex linear relationships between the variables, but is much costlier to perform.

- Various numerical domains populate the space between intervals and convex polyhedra, allowing a trade-off between precision and speed. For example, [CC04] explores the use of octahedral constraints, which are polyhedra where the half-spaces have the restricted form $\pm x_1 \pm \ldots \pm x_n \leq a$. Here $\sqcup$ computes the so-called octahedral hull.

In addition to finding linear constraints, some progress has been made with polynomial constraints, [RCK04] being a particularly striking example. That work begins by observing that the set of polynomials which are always zero at a given program point (equivalently, the set of polynomial invariants at that point) forms an *ideal* in the ring of polynomials, and therefore can be finitely represented as a Gröbner basis. Furthermore, intersection of ideals can be computed using Gröbner bases, as can the effect of instructions (for a restricted class of programs). Thus, Gröbner bases representing sets of polynomials are used as the abstract values, and the $f(i)^\#$ and $\sqcup$ functions are implemented using an existing library for manipulating such bases. A widening operator is also given, which ensures convergence by bounding the degree of invariants found.

As a final remark, we note that constraints such as polyhedra can be encoded into a first order logic with arithmetic, but the specialised representations that have been developed (e.g. the *octahedron decision diagrams* of [CC04]) and related specialised algorithms argue against doing this in practice; treating e.g. octahedra simply as first order formulae obscures the structure which makes the specialised algorithms work. The same applies to many abstract domains, and this relates to benefit B2 in Section 1.2.2.

2.6 Abstraction domains are not independent

We conclude our background chapter with a discussion of the non-independence of abstraction domains. We begin with an example of this non-independence.

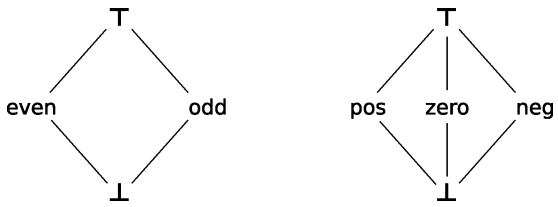

Figure 2.11: The parity and sign abstraction lattices

2.6.1 An example of non-independence

Example 2.6.1. Figure 2.11 shows two abstraction domains for a program with a single integer variable: the sign and parity domains, which we call **Sgn** and **Par** . These domains record only the sign (resp. parity) that an integer variable has; their concretisation functions are as follows:

$$\begin{aligned}
\mathbf{Sgn}.\gamma(\text{pos}) &= \{n \in \mathbb{Z} : n > 0\} \\
\mathbf{Sgn}.\gamma(\text{zero}) &= \{0\} \\
\mathbf{Sgn}.\gamma(\text{neg}) &= \{n \in \mathbb{Z} : n < 0\} \\
\mathbf{Sgn}.\gamma(\top) &= \mathbb{Z} \\
\mathbf{Sgn}.\gamma(\bot) &= \emptyset
\end{aligned}$$

$$\begin{aligned}
\mathbf{Par}.\gamma(\text{even}) &= \{n \in \mathbb{Z} : \exists m \in \mathbb{Z}.n = 2m\} \\
\mathbf{Par}.\gamma(\text{odd}) &= \{n \in \mathbb{Z} : \exists m \in \mathbb{Z}.n = 2m + 1\} \\
\mathbf{Par}.\gamma(\top) &= \mathbb{Z} \\
\mathbf{Par}.\gamma(\bot) &= \emptyset
\end{aligned}$$

Suppose we have the abstract values even and pos for these domains, respectively, before execution of the instruction `x := x - 1`. Run by itself, the parity domain produces

$$\mathbf{Par}.f(x := x - 1)^{\#}(\text{even}) = \text{odd}$$

and, run by itself, the sign domain produces

$$\mathbf{Sgn}.f(x := x - 1)^{\#}(\text{pos}) = \top$$

because the sign of the new value of `x` cannot be determined.

But if we can combine the information from the two domains, rather than just invoking their successor functions pointwise, we can do better: we know that if an integer n is even and $n > 0$, then $n \geq 2$. Therefore $n - 1$ must be positive and odd, and the more precise pair of answers odd and pos is preferable. □

2.6.2 Reduced product: a non-algorithmic description of domain combination

The fact that more precise results can be obtained by combining the information in two domains, rather than invoking their successor functions pointwise, is discussed in [CC79]. That work describes a *reduced product operator* for combining domains.

Suppose we have two abstraction domains $\mathbf{D}$ and $\mathbf{E}$. To get the reduced product $\mathbf{D} \otimes \mathbf{E}$, we take the set of abstract values to be

$$(\mathbf{D} \otimes \mathbf{E}).A \quad := \quad \mathbf{D}.A \times \mathbf{E}.A$$

i.e. an abstract value in the reduced product $\mathbf{D} \otimes \mathbf{E}$ is a pair consisting of one abstract value from $\mathbf{D}$ and one from $\mathbf{E}$. The transfer functions are then defined by:

$$(\mathbf{D}\otimes\mathbf{E}).f(i)^{\#}(a,b) \quad := \quad (\mathbf{D}.\alpha(X), \mathbf{E}.\alpha(X)) \quad \text{where} \quad X = f(i)(\mathbf{D}.\gamma(a) \cap \mathbf{E}.\gamma(b))$$

(Recall that the abstraction function α, described on page 43, maps a set of states to its unique best abstract representation, which in the setting used in [CC79] is guar-

anteed to exist.) Informally, the preceding definition says that we should concretise the two abstract values, intersect their concretisations, apply the *concrete* transfer function for the statement, and then choose the best abstract value to represent the result in each of the domains.

By way of example, let us apply this definition to Example 2.6.1: we wish to calculate the successor under `x := x-1` of $(\text{pos}, \text{even})$ in the reduced product $\mathbf{Sgn} \otimes \mathbf{Par}$ i.e.

$$(\mathbf{Sgn} \otimes \mathbf{Par}).f(x := x - 1)^{\#}(\text{pos}, \text{even})$$

We first calculate (informally) the common subexpression

$$\begin{aligned} & f(x := x - 1)(\mathbf{Sgn}.\gamma(\text{pos}) \cap \mathbf{Par}.\gamma(\text{even})) \\ = \; & f(x := x - 1)(\{1, 2, 3, 4, 5, 6, \ldots\} \cap \{\ldots, -6, -4, -2, 0, 2, 4, 6, \ldots\}) \\ = \; & f(x := x - 1)(\{2, 4, 6, \ldots\}) \\ = \; & \{1, 3, 5, \ldots\} \end{aligned}$$

and then we use the reduced product definition:

$$\begin{aligned} & (\mathbf{Sgn} \otimes \mathbf{Par}).f(x := x - 1)^{\#}(\text{pos}, \text{even}) \\ = \; & (\mathbf{Sgn}.\alpha(\{1, 3, 5, \ldots\}), \mathbf{Par}.\alpha(\{1, 3, 5, \ldots\}) \\ = \; & (\text{pos}, \text{odd}) \end{aligned}$$

which gives the more precise answer as desired. In this sense, the reduced product is the "right" way to combine domains. The reduced product operator can be iterated to combine any number of domains.

Unfortunately, however, there is a big problem: *the definition of reduced product is non-algorithmic*. This means that the reduced product does not tell us *how to implement* domain combination at all: it *describes* non-algorithmically what the combined domain looks like, but doesn't give us a recipe for implementing it.

Figure 2.12: The reduced product is really a non-algorithmic *description* of ideal domain combination; it is *not* an effective way of realising domain combination. Describing something one would like, in this case domain combination, doesn't produce that thing (assuming one is not allowed to invoke the power of Santa Claus).

To see this, observe that the definition of the transfer functions for the product domain is written in terms of operations involving sets of program states, such as $f(i)$ and $\cap$; the reason we introduced abstraction domains in the first place is precisely that we cannot compute with such sets!

If we are labouring this point, that is only because the point seems not to be widely understood: while carrying out this work we have frequently been asked "Why don't you just use the reduced product?" as if that settled the matter of domain combination. But what does it mean to "use the reduced product" when it is non-algorithmic? The reduced product can serve as a *description* of ideal domain combination, but it doesn't give us an effective way of realising that combination. And as shown in Figure 2.12[3], merely describing something one would like doesn't produce that thing.

[3]The clipart Santa is from ToonWorkshop.com and the clipart boy is from Free-Clipart-Pictures.net. Both are free for non-commercial use. The composition is my own.

	$\top$	even	odd	$\bot$
$\top$	$(\top, \top)$	($\top$, odd)	($\top$, even)	·
pos	$(\top, \top)$	(pos, odd)	($\top$, even)	·
zero	·	(neg, odd)	·	·
neg	(neg, $\top$)	(neg, odd)	(neg, even)	·
$\bot$	·	·	·	$(\bot, \bot)$

Figure 2.13: Direct implementation of the abstract transfer function for `x:=x-1` in the product domain **Sgn**⊗**Par**. Where the entry for a pair is missing, this is because that pair should never be encountered during the analysis.

2.6.3 Direct implementation of combined domains

Of course, there is nothing to stop us manually implementing abstract transfer functions for a particular product domain: these can then combine the information from the various domains and thus achieve greater precision as above. Figure 2.13 gives a direct implementation of the transfer function for `x:=x-1` in the product domain $\mathbf{Sgn} \otimes \mathbf{Par}$.

In some cases, such as for the product of the sign and parity abstractions, we can make a direct implementation that matches the precision of the reduced product. In other cases this is impossible because the transfer functions described by the reduced product are not computable; nevertheless a direct implementation can still do better than pointwise application (and its design may still be informed by the reduced product).

We mention in passing that in such product domains, we can expect to find pairs that should never be generated by the transfer functions, and so can be ignored. Here, for instance, the pair (zero, $\top$) shouldn't be generated, because zero is even; the pair (zero, even) is always better and should be generated instead. Similarly the pair (zero, odd) is inconsistent (has an empty concretisation); $(\bot, \bot)$ should be used instead.

Direct implementation of product domains has the following disadvantages:

- The interactions between domains can be subtle and hence a direct implemen-

tation can be time-consuming and difficult to get right.

- Direct implementation is non-modular: the implementor needs to understand the structure of all the domains involved, of which there may in general be any number.
- Direct implementations are less reusable: a direct implementation must be redone each time we want to change the set of domains used.

As we explained in Section 1.2.2 of the Introduction, in this thesis we aim for clean combination of separately-implemented domains, which makes implementation modular and thus more manageable (benefit B3), and means that different abstraction domains can be mixed and matched (benefit B4).

2.6.4 Modular combination: the open product operator

When we began this work, the only thing we could find in the literature concerning modular combination of domains was the *open product operator* [CCH00][4], which we therefore took as our starting point. We now explain the open product in informal, simplified terms.

The idea of the open product operator is that the domains can send each other queries about program state. So one begins by fixing a set *Query* of queries about program state.

The signature of each abstraction domain is then enlarged to include a computable (Curried) "query-answering function"

$$answer : A \to Query \to \{true, unknown\}$$

the idea being that $answer(a)(q)$ evaluates the query q on the set of states represented by a, returning *true* if the query is true in all such states.

[4]We thank Dennis Dams for first pointing out to us the existence of the open product operator.

The "type" of the abstract transfer functions is changed too: instead of $f(i)^{\#} : A \to A$ it becomes

$$f(i)^{\#} : A \times (Query \to \{true, unknown\}) \to A$$

so that each abstract transfer function takes a query-answering function as an extra argument, which it can then interrogate.

The abstract transfer functions on the product domain $\mathbf{D} \otimes \mathbf{E}$ are defined by

$$(\mathbf{D}\otimes\mathbf{E}).f(i)^{\#}(a,b) \quad := \quad \big(\quad \mathbf{D}.f(i)^{\#}(a, \mathbf{E}.answer(b)), \quad \mathbf{E}.f(i)^{\#}(b, \mathbf{D}.answer(a)) \quad \big)$$

In other words, the abstract transfer function from each of the domains is given the ability to interrogate the query-answering function from the other domain.

If in Example 2.6.1 the query set Q included a query $x \neq 1$ then the sign analysis could send this query to the parity analysis, and we would get the more precise answer (pos, odd) as desired.

In [CCH00] the open product is applied to the detection of shallow properties of Prolog programs, for the purposes of optimisation. In this thesis we rework the idea in the context of verifying deep behavioural specifications of imperative heap-manipulating programs.

2.6.5 Comparison: open product vs. reduced product

Like the direct implementation method, the open product operator can be used to obtain transfer functions matching those described by the reduced product, in cases where the latter are computable; in other cases it can be used as a "good" approximation of the reduced product, giving transfer functions that are more precise than pointwise application. Unlike the direct implementation method, the open product operator is modular.

The central issue, when discussing the open product and comparing it with the reduced product, is the choice of query language Q. This is because the domains being combined with the open product can share information *only to the extent that is allowed by the query language*. If the query language chosen is not expressive enough, then the domains will not be able to communicate the required information to each other, and the transfer functions produced will be less precise than desired; in particular, they will not match those of the reduced product even when the latter are computable. In the extreme case, where one chooses the empty query language $Q = \emptyset$, the open product degenerates into pointwise application. On the other hand, as one increases the expressiveness of the query language, the amount of programming required to implement each domain increases, because more complicated queries must be answered and issued. Thus the choice of query language will be a significant consideration in this thesis.

2.7 Summary

In this chapter we introduced the problem of automatically verifying software, and saw that this is difficult because most software programs are infinite-state. We introduced the concepts of inductive verification and abstraction, which overcome the infiniteness of the state space by using *abstract states* to represent *sets* of program states. These two concepts are the basis for the automated software verification systems in use today.

We then surveyed the abstraction domains currently in use, introducing among others domains based on predicate abstraction, shape analysis and numerical constraints. We noted that each domain has its particular strengths and weaknesses, in terms of: the kind of properties discovered, the precision of the analysis, the classes of programs handled and the degree of automation. It is precisely because of these varying strengths and weaknesses that we wish to provide for cooperation between

diverse verification techniques.

We next observed that running the various domains independently on the same target program is not sufficient: if we are to obtain precise results, the domains have to share their information. We explained that although there is an operator, the *reduced product*, which gives a *description* of ideal domain combination, this operator cannot be implemented; its definition is non-algorithmic and sometimes describes transfer functions that are not computable.

Finally, we unearthed the *open product* operator from the literature, which allows domains to exchange information by sending each other queries about the target program's states. The choice of the query language is important because the domains being combined can cooperate only to the extent that is allowed by the query language. Open products had been used for the shallow optimisation of Prolog programs, but not yet applied to verification. Unlike the reduced product, this operator can be used for modular automatic combination. In the next chapter, we will use the ideas of the open product to design and formalise a software verification system in which the various verification techniques are implemented independently, but automatically cooperate to verify target programs.

Chapter 3

Our approach: basic concepts and algorithms

In this chapter we present and formalise the fundamentals of our new verification method.

- We begin by describing and then formalising our choice of **programming language** for our verification methods to target: we choose an idealised imperative heap-manipulating language with non-deterministic choice and recursive procedures, represented as CFGs. We define program states for this language, and give operational semantics.
- Next we present a logic over program states, which will serve as the **single common language** in which the analysis modules exchange information; we use a first order logic extended with transitive closure.
- We then introduce the notion of an **abstract model of a program**, and say what it means for such a model to be sound (with respect to the operational semantics). We show how sound models can be used to prove the non-reachability of "bad" CFG nodes, and thus verify safety properties of programs.

- We set out our notion of an **analysis module**, which is central to our work. Analysis modules implement a common interface, which extends that of an abstraction domain, adding functions for the propagation of information between modules. We state soundness conditions which analysis modules must meet.

- We give an **algorithm for verification**, which is generic in that it works with any sound analysis module. This algorithm is worklist-based and uses summarisation to handle recursive procedure calls without requiring that procedures be annotated with pre- and post-conditions. We prove that the algorithm terminates and produces correct results.

- Finally we define our **combination operator** $\diamond$, which combines two analysis modules in such a way that they work together cooperatively. We show that the combination of two sound analysis modules is again a sound analysis module. Thus we can use our verification algorithm with any number of cooperating analysis modules.

3.1 The target programming language

Our verification method targets an idealised imperative language with recursive procedures. There are statements to allocate and manipulate heap objects, including block allocation, allowing for linked data structures and arrays. The language also includes non-deterministic choice. On the other hand the language does not include facilities for exceptions, inheritance or concurrency, all of which introduce subtle difficulties. Extending our techniques to account for these is well beyond the scope of a (one-man) PhD project which produces an implementation, and thus is left as future work. Also, allocation in our language is irreversible: once a block of memory is allocated it can be neither garbage-collected nor explicitly deallocated.

We now present the language and its semantics. Before plunging into definitions, here is a brief summary and explanation of what follows:

- Each program consists of declarations for a number of heap fields and then a number of procedures. Procedure bodies are represented by control flow graphs, as in Section 2.1.1, and may be recursive.
- Statements labelling edges of the CFGs must take very simple forms, such as $x := a + b$.
- The language is essentially untyped: all variables hold integers, which can be used either for arithmetic or as the addresses of heap objects.
- Each program state has the form (e, h, A) where e is the environment (mapping local variables to their values), h is the heap (mapping address-field pairs to their values) and A is the allocation set (the subset of heap addresses that are allocated).

3.1.1 Syntax of programs

Fix once and for all countably infinite disjoint sets *Vars*, *Fields* and *ProcNames*. Also fix once and for all a countably infinite set of control locations *Locs* with designated distinct elements *start*, *memerror*, *asserterror* and *terminated*.

Definition 3.1.1. A *control flow graph*, or CFG, $G = (N, E)$ consists of:

- A finite set of nodes $N \subseteq Locs$, which includes three special nodes, with the following meanings:
  - *start*: denotes that the procedure begins here
  - *memerror*: represents a memory safety error
  - *asserterror*: represents an assertion violation

and may also include the special node

  - *terminated*: represents normal termination of the program

- An edge function $E : N \to \mathit{DStmt}$, where the set *DStmt* of "directed statements" contains the following kinds of elements (the comments on the right show how these statements might be written in source code):

`Skip` : n	// do nothing
`VarCopy`(u, v) : n	u := v
`AssignConst`(u, k) : n	u := k
`Arith`$(u, v, \otimes, v')$: n	u := $v \oplus v'$
`FieldRead`(u, v, f) : n	u := v.f
`FieldWrite`(v, f, u) : n	v.f := u
`New`(u, v) : n	u := new array[0..v-1]
`Call`$(u, \pi, [p_1, \ldots, p_k])$: n	u := $\pi(p_1, \ldots, p_k)$
`Return`(v)	return v
`If`(Φ) : n : n'	if (Φ) then ... else ...
`Choice` : n : n'	// nondeterministic choice

where $u, v, v' \in \mathit{Vars}$, $k \in \mathbb{Z}$, $\otimes \in \{+, -, \times\}$, $\pi \in \mathit{ProcNames}$, $f \in \mathit{Fields}$ and Φ is a guard formula belonging to the logic $\mathscr{L}^{\{\mathfrak{O},\mathfrak{C}\}}$ whose introduction we defer until later.

The node component $n \in N$ in the above statement forms indicates which CFG node control passes to after execution of the statement. For the `If` and `Choice` statements (which, strictly speaking, label hyperedges rather than edges) there is a choice between two nodes, so these statement forms have two components $n, n' \in N$.

In a control flow graph there may be at most one node labelled with a `Return` edge; this node is called the return point.

The special nodes other than *start*, i.e. *memerror*, *asserterror* and *terminated* (if present), must each have a self-loop formed with a `Skip`-edge.

We will use appropriately named projection functions when dealing with tuples: here the functions *Nodes* and *Edges* will project to the respective components of a CFG. □

The different types of edges encode different program statements; we work with a minimal set of statements in order to reduce the number of cases our analysis modules must handle. Thus, more complicated statements appearing in source code, such as assignments with multiple arithmetic operators on the right hand side, must be represented in the CFG by a sequence of simple statements, possibly using temporary variables.

The nondeterministic choice statement `Choice` is used when we want to say that both choices are possible. This can be used to model choices which are in the control of the environment (for example user input), and to create "generators" when complex data structures are needed as inputs (we shall see this in Chapter 5). For example, to test a program that operates on acyclic linked lists, we would prepend to it a call to a procedure which nondeterministically generates all possible acyclic linked lists. (This approach is used by others e.g. in [BCC+07] and is related to the idea in model checking that actions controlled by the environment are treated as nondeterministic, to cover all possible cases.) In Chapter 6 we will see that, from the point of view of *falsifying* a program, a `Choice` statement can be given a stronger treatment than an `If` statement which we "cannot resolve": for `Choice` both choices are possible, whereas for an unresolved `If` exactly one choice happens but we cannot tell which.

Definition 3.1.2. A *procedure* is a 4-tuple $\Pi = (\pi, [p_1, \ldots, p_j], [l_1, \ldots, l_k], G)$ where: $\pi \in$ *ProcNames* is the procedure's name, $p_i \in$ *Vars* are the formal parameters, $l_i \in$ *Vars* are its local variables (disjoint from the p_is) and G is its control flow

graph. The functions *Name*, *Formals*, *Locals* and *Graph* will project to the respective components of a procedure. □

Definition 3.1.3. A *program* P is a pair $([f_1, \ldots, f_m], [\Pi_1, \ldots, \Pi_n])$ where $f_i \in$ *Fields* are the heap fields used by that program, and Π_i are its procedures (with Π_1 taken to be the program's "main" procedure, where execution starts), such that the following "healthiness conditions" hold:

- Whenever a variable $v \in$ *Vars* is mentioned by an edge in the CFG of procedure Π_i, v occurs either in *Formals*(Π_i) or in *Locals*(Π_i).
- For every edge $\texttt{FieldRead}(u, v, f) : n$ and every edge $\texttt{FieldWrite}(v, f, u) : n$ occurring in the CFGs, f occurs in the list $[f_1, \ldots, f_m]$ of fields.
- For every edge $\texttt{Call}(u, \pi, [p_1, \ldots, p_k]) : n$ occurring in the CFGs, there exists a procedure Π_j in the program such that *Name*$(\Pi_j) = \pi$ and *Formals*(Π_j) has length k.
- For each procedure Π_i in the program, the formal parameter list *Formals*(Π_i) does not contain duplicate elements.
- The program does not contain two distinct procedures Π_i and Π_j such that *Name*$(\Pi_i) =$ *Name*(Π_j).
- The main procedure Π_1 has an empty parameter list, does not have a return point, and there are no calls anywhere to the main procedure. The main procedure includes the node *terminated* and no other procedure does.

Since procedures are uniquely named in programs, we shall afford ourselves the liberty of writing π when we really mean "the procedure Π whose name is π", and vice versa. The respective projection functions will be called *Procs* and *Fields*. □

Running example:

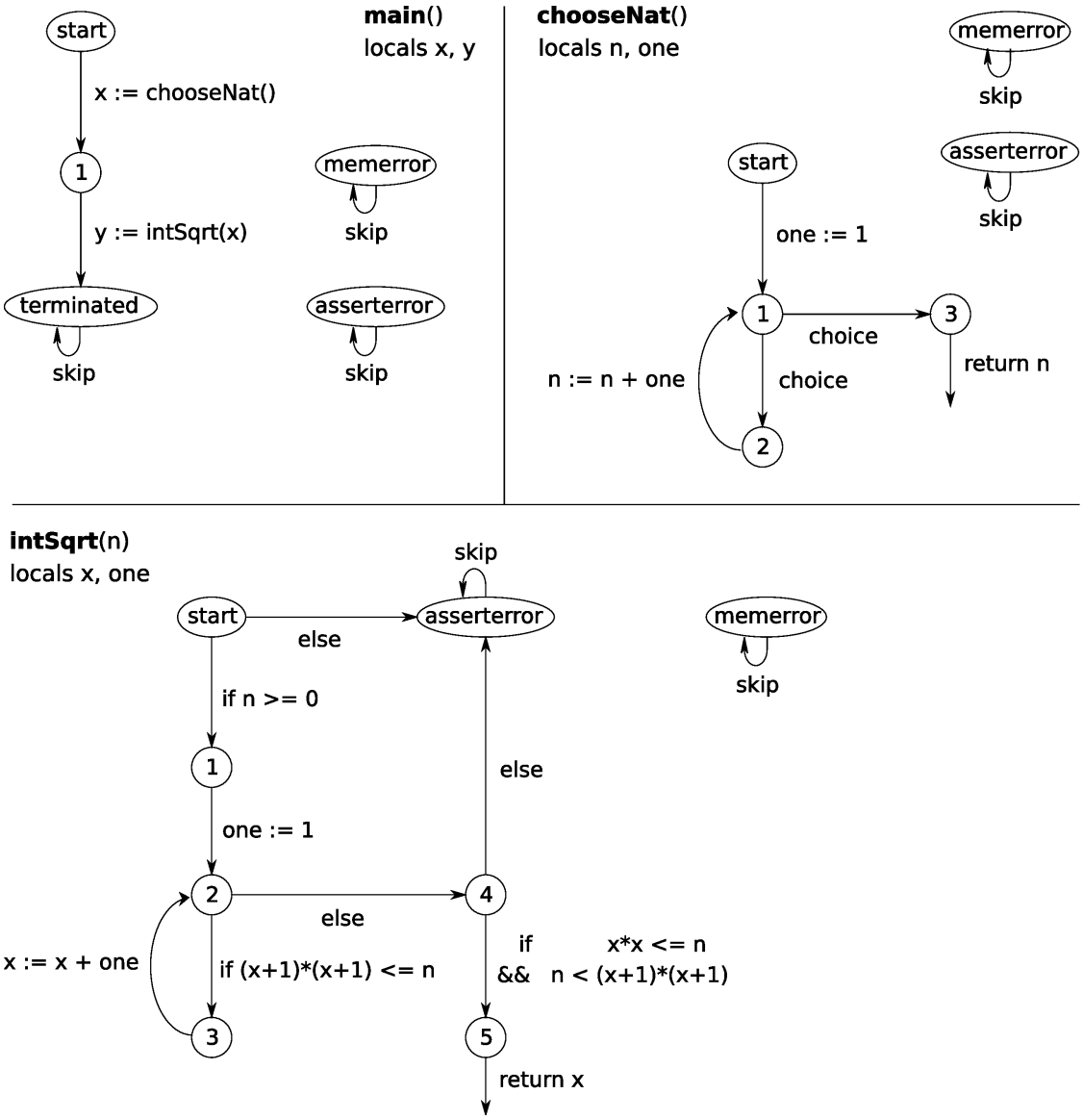

Figure 3.1: CFGs for our running example program.

To illustrate the above we show a simple example program, consisting of three procedures called *main*, *chooseNat* and *intSqrt*. The CFGs are in Figure 3.1.

The *chooseNat* procedure nondeterministically returns *all* natural numbers, using the `Choice` statement. The *main* procedure calls *chooseNat* and then uses the result as the input to *intSqrt*, which calculates integer square root as in the previous chapter. This arrangement ensures that *intSqrt* is checked on *all* its legal inputs. Note the introduction of the variable *one*; this is because the statement `x := x+1` used in the previous chapter is not in our minimal set of statements.

We can (if we really want) write these CFGs down formally, so that e.g. for *chooseNat* we have $G_{chooseNat} = (N, E)$ where

$$N = \{start, asserterror, memerror, 1, 2, 3\}$$

and

$$\begin{aligned} E(start) &= \texttt{AssignConst}(one, 1) : 1 \\ E(1) &= \texttt{Choice} : 2 : 3 \\ E(2) &= \texttt{Arith}(n, n, +, one) : 1 \\ E(3) &= \texttt{Return}(n) \end{aligned}$$

The *chooseNat* procedure is then formally $\Pi_2 = (chooseNat, [], [n, one], G_{chooseNat})$ (recall that the components are the name, formal parameter list, local variable list and CFG respectively). Finally the whole program is

$$\begin{aligned} &([], [\\ &\quad (main, [], [x, y], G_{main}), \\ &\quad (chooseNat, [], [n, one], G_{chooseNat}), \\ &\quad (intSqrt, [n], [x, one], G_{intSqrt}) \\ &\quad]) \end{aligned}$$

(recall that the components are the list of heap fields used, and the list of procedures).

3.1.2 Program states

We now define a notion of *program state*. This is the first step in giving semantics to our programs.

Definition 3.1.4. An *environment* is a function $Vars \to \mathbb{Z}$. Let *Env* be the set of all environments. □

Definition 3.1.5. A *heap* is a function from $Fields \times \mathbb{Z}$ to $\mathbb{Z}$. We use 0 as the null address (and will use 0 and *null* interchangeably from now on). Let *Heap* be the set of all heaps. □

Definition 3.1.6. An *allocation set* A is a subset $A \subseteq \mathbb{Z}_{>0}$ used to record which heap locations have been allocated. Let *AllocSet* be the set of all allocation sets.

Definition 3.1.7. Finally, a *program state* is a triple (env, h, A) where *env* is an environment, h is a heap and A is an allocation set. Let *State* be the set of all program states. We name the respective projections *Env*, *Heap* and *AllocSet*. □

We also define a special *starting state*, in which programs will begin their execution.

Definition 3.1.8. The *starting state*, denoted s^{start}, is the triple (env, h, A) where: *env* and h have value 0 everywhere, and $A = \emptyset$. □

3.1.3 Semantics of programs

Now we give operational semantics to our programs. In our background chapter we did this by associating with each statement i a transfer function $f(i) : State \to \mathbb{P}(State)$. From this we then defined a transition relation $semantics(P) \subseteq S_{loc} \times S_{loc}$ on located states, and finally the set of reachable located states $reach(P)$.

Here, however, our formalisation is slightly different. Note that even though our language has procedures, our states do not contain calls stacks; using explicit call stacks would not be a good fit with procedure summarisation. Instead, we track the call stack implicitly in the operational semantics.

Our semantics consists of two judgements, one for intraprocedural execution and the other for procedure calls:

A. Intraprocedural execution judgement:

$$\pi, s_0 \xrightarrow{p} l, s$$

This can be read as: "Execution can reach location l in procedure π, with state s, and the state when the procedure was entered was s_0". (The role of p is discussed shortly.)

B. Procedure call judgement:

$$\pi, s_0 \rightarrow l, s : \pi', s'_0$$

This can be read as: "Execution can reach location l in procedure π, with state s, and the state when the procedure was entered was s_0. Then the procedure π' is called, and after parameter passing the state is s'_0".

The coming definitions of these judgements will be mutually dependent. In one direction, procedure calls happen when intraprocedural execution reaches a `Call` edge: in this case we derive judgements of type B (procedure call) from those of type A (intraprocedural execution). On the other hand, to start the called procedure running or to restart the caller upon return, we derive judgements of type A (intraprocedural execution) from those of type B (procedure call).

The value p in the intraprocedural execution judgement gives us an indication of what the previous step in the execution was (but not a full trace), and will be useful when we give the soundness conditions for abstract models (in Definition 3.3.2).

Each p and can take the following forms:

$p = \epsilon$**:** This means we are at the beginning of a procedure, so there is no previous step of execution.

$p = l', s'$**:** This means the previous step of execution was the running of the intraprocedural statement at node l' in the state s'.

$p = l_1, s_1, \pi', s_2, l_3, s_3$**:** The means the previous step of execution was a call, at location l_1 and in state s_1, to the procedure π'. The state upon entering the called procedure (i.e. after parameter-passing) was s_2, and then execution flowed to location l_3 and state s_3 where it returned to the caller.

Figures 3.2, 3.3, 3.4 and 3.5 give the (named) derivation rules for the two judgements. Strictly speaking, we mean to use the smallest pair of relations which is closed under these rules.

The rules in Figures 3.2, 3.3 and 3.4 involve only judgement A (intraprocedural execution) and thus we call them *intraprocedural rules*. The init rule is used to start the program's main procedure executing in the initial state. The other rules each, in effect, extend execution one further step through the program. Consider for example the varcopy rule:

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{VarCopy}(u, v) : l' \\ s = (e, h, A) \\ s' = (e', h, A) \\ e' = e \oplus \{u \mapsto e(v)\}\end{array}}{\pi, s_0 \xrightarrow{l,s} l', s'}$$

Informally the premises mean:

1: Execution can reach location l in procedure π, with state s, and the state when the procedure was entered was s_0.

2: The CFG for procedure π contains an edge labelled $\texttt{VarCopy}(u, v)$ from location l to location l'.

3, 4, 5: The state s' is the same as s except that the environment has been updated at variable u, which is now mapped to the existing value of variable v. (Throughout, $\oplus$ denotes the function override operator.)

The conclusion extends execution one step: execution can reach location l' in π, with new state s' and the state when the procedure was entered was s_0. The value above the arrow records that the previous step of execution was the running of the intraprocedural statement at node l in the state s.

These rules are mostly straightforward. Conditions on `If` edges are evaluated using $[\![\Phi]\!]$ which denotes (roughly) the set of program states in which Φ holds; this is defined in the next section. The statements for potentially dangerous memory accesses (field read and field write) have two rules, one for successful execution and the other for failure. Failure occurs when the address accessed is not allocated. The failure rules transfer execution straight to the special node *memerror*.

The block allocation statement `New` also has separate success and failure rules: allocation fails if a memory block of non-positive length is requested. The rule for successful allocation is nondeterministic: the block can be allocated anywhere it fits.

Figure 3.5 gives the rules for procedure calls and returns. These rules mix the two judgement forms A (intraprocedural execution) and B (procedure call), which are mutually dependent. According to the call-1 rule, local variables are initialised to zero, and parameter passing is done in the call-by-value style. The procedure call process is split into two stages, call-1 and call-2, because we do not use explicit stacks: the calling context is recorded in the conclusion of the call-1 rule so that it can be restored upon return.

It is worth remarking on the semantics of the nondeterministic `Choice` statement. Whenever such a statement is encountered, both the rules choice-1 and choice-2 will be applicable: the current execution will extend in *two ways* corresponding to the two branches. Of course, in *any particular execution* exactly one of the choices is

init

$$\frac{}{\pi_1, s^{start} \xrightarrow{\epsilon} start, s^{start}}$$

skip

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Skip} : l'\end{array}}{\pi, s_0 \xrightarrow{l,s} l', s}$$

varcopy

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{VarCopy}(u, v) : l' \\ s = (e, h, A) \\ s' = (e', h, A) \\ e' = e \oplus \{u \mapsto e(v)\}\end{array}}{\pi, s_0 \xrightarrow{l,s} l', s'}$$

assignconst

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{AssignConst}(u, k) : l' \\ s = (e, h, A) \\ s' = (e', h, A) \\ e' = e \oplus \{u \mapsto k\}\end{array}}{\pi, s_0 \xrightarrow{l,s} l', s'}$$

arith

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Arith}(u, v_1, \otimes, v_2) : l' \\ s = (e, h, A) \\ s' = (e', h, A) \\ e' = e \oplus \{u \mapsto e(v_1) \otimes e(v_2)\}\end{array}}{\pi, s_0 \xrightarrow{l,s} l', s'}$$

fieldread-success

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{FieldRead}(u, v, f) : l' \\ s = (e, h, A) \\ s' = (e', h, A) \\ e(v) \in A \\ e' = e \oplus \{u \mapsto h(f, e(v))\}\end{array}}{\pi, s_0 \xrightarrow{l,s} l', s'}$$

Figure 3.2: Derivation rules for intraprocedural execution. These rules involve only judgement A (intraprocedural execution). Part 1 of 3.

fieldread-failure

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{FieldRead}(u, v, f) : l' \\ s = (e, h, A) \\ e(v) \notin A\end{array}}{\pi, s_0 \xrightarrow{l,s} \mathit{memerror}, s}$$

fieldwrite-success

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{FieldWrite}(v, f, u) : l' \\ s = (e, h, A) \\ s' = (e, h', A) \\ e(v) \in A \\ h' = h \oplus \{(f, e(v)) \mapsto e(u)\}\end{array}}{\pi, s_0 \xrightarrow{l,s} l', s'}$$

fieldwrite-failure

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{FieldWrite}(v, f, u) : l' \\ s = (e, h, A) \\ e(v) \notin A\end{array}}{\pi, s_0 \xrightarrow{l,s} \mathit{memerror}, s}$$

new-success

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{New}(u, v) : l' \\ s = (e, h, A) \\ s' = (e', h, A') \\ e(v) > 0 \\ a > 0 \\ \{a, a+1, \ldots, a+e(v)-1\} \cap A = \emptyset \\ A' = A \cup \{a, a+1, \ldots, a+e(v)-1\} \\ e' = e \oplus \{u \mapsto a\}\end{array}}{\pi, s_0 \xrightarrow{l,s} l', s'}$$

new-failure

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{New}(u, v) : l' \\ s = (e, h, A) \\ e(v) \leq 0\end{array}}{\pi, s_0 \xrightarrow{l,s} \mathit{memerror}, s}$$

Figure 3.3: Derivation rules for intraprocedural execution. These rules involve only judgement A (intraprocedural execution). Part 2 of 3.

choice-1

$$\frac{\pi, s_0 \xrightarrow{p} l, s \quad \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Choice} : l'_1 : l'_2}{\pi, s_0 \xrightarrow{l,s} l'_1, s}$$

choice-2

$$\frac{\pi, s_0 \xrightarrow{p} l, s \quad \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Choice} : l'_1 : l'_2}{\pi, s_0 \xrightarrow{l,s} l'_2, s}$$

if-true

$$\frac{\pi, s_0 \xrightarrow{p} l, s \quad \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{If}(\Phi) : l'_1 : l'_2 \quad (s_0, s) \in [\![\Phi]\!]^{\{\mathfrak{O},\mathfrak{C}\}}}{\pi, s_0 \xrightarrow{l,s} l'_1, s}$$

if-false

$$\frac{\pi, s_0 \xrightarrow{p} l, s \quad \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{If}(\Phi) : l'_1 : l'_2 \quad (s_0, s) \notin [\![\Phi]\!]^{\{\mathfrak{O},\mathfrak{C}\}}}{\pi, s_0 \xrightarrow{l,s} l'_2, s}$$

Figure 3.4: Derivation rules for intraprocedural execution. These rules involve only judgement A (intraprocedural execution). Part 3 of 3.

call-1

$$\frac{\begin{array}{c}\pi, s_0 \xrightarrow{p} l, s \\ \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Call}(u, \pi', [p_1, \ldots, p_k]) : l' \\ s = (e, h, A) \\ s' = (e', h, A) \\ [f_1, \ldots, f_j] = \mathit{Formals}(\pi') \\ e'(x) = \begin{cases} e(p_i) & \text{if } x \text{ is } f_i \\ 0 & \text{otherwise} \end{cases}\end{array}}{\pi, s_0 \rightarrow l, s : \pi', s'}$$

call-2

$$\frac{\pi, s_0 \rightarrow l, s : \pi', s'}{\pi', s' \xrightarrow{\epsilon} \mathit{start}, s'}$$

return

$$\frac{\begin{array}{c}\pi, s_0 \rightarrow l_1, s_1 : \pi', s_2 \\ \pi', s_2 \xrightarrow{p} l_3, s_3 \\ \mathit{Edges}(\mathit{Graph}(\pi))(l_1) = \texttt{Call}(u, \pi', [p_1, \ldots, p_k]) : l_c \\ \mathit{Edges}(\mathit{Graph}(\pi'))(l_3) = \texttt{Return}(v) \\ s_1 = (e_1, h_1, A_1) \\ s_3 = (e_3, h_3, A_3) \\ s_c = (e_c, h_3, A_3) \\ e_c = e_1 \oplus \{u \mapsto e_3(v)\}\end{array}}{\pi, s_0 \xrightarrow{l_1, s_1, \pi', s_2, l_3, s_3} l_c, s_c}$$

Figure 3.5: Derivation rules for procedure calls and returns. These rules mix the two judgement forms A (intraprocedural execution) and B (procedure call) which are mutually dependent.

taken, but for both branches there *exists* an execution which follows that branch.

Consider for instance the procedure *chooseNat* in Figure 3.1. In *any particular execution* this procedure returns a single natural number; but for every number $n \in \mathbb{N}$ there *exists* an execution which returns n. This is what we meant when we said that "*chooseNat* returns all natural numbers".

Remark 3.1.9. Using this semantics, a CFG node l in a procedure π is reachable iff there exist states s_0 and s such that (for some p) $\pi, s_0 \xrightarrow{p} l, s$.

3.2 Our logic for program states

In this section we present the logic $\mathscr{L}$ which is used to describe program states. In fact, $\mathscr{L}$ will do triple duty, being used to express:

- guards for alternation and iteration statements in programs,
- assertions about desired program behaviour and
- information exchanged between cooperating analysis modules.

But before we give syntax and semantics for $\mathscr{L}$, we must introduce the idea of time indices.

3.2.1 The need for time indices

If we are to use our logic to describe the effect of a program statement, we will need a way of relating the program states before and after the statement's execution. A standard way to do this (as in e.g. [Mor94]) is to add a subscript, here $\mathbf{I}$, to variables which should be evaluated in the "before" state. Hence, for example, if the variables

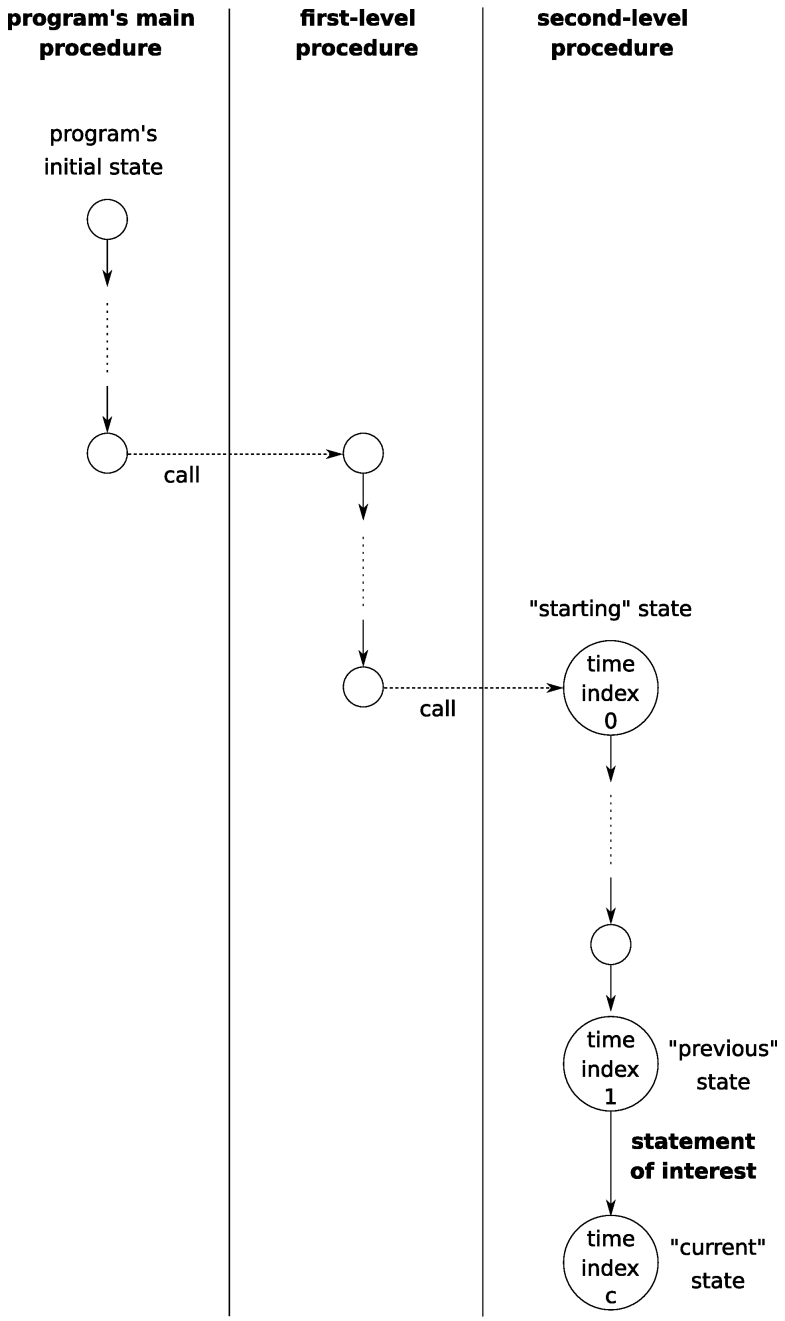

Figure 3.6: An illustration of the roles of the time indices $\mathfrak{O}$, $\mathfrak{I}$ and $\mathfrak{C}$ when considering the execution of an intraprocedural statement.

in scope are x and y, the statement `x := x+1` can be described by the formula

$$x = x_{\mathfrak{I}} + 1 \quad \wedge \quad y = y_{\mathfrak{I}}$$

We will also need to write "frame conditions" which say that, for instance, certain variables have not changed since the beginning of a procedure's execution. We decorate variables with the subscript $\mathfrak{0}$ to indicate that they should be evaluated in the program state as it was on entry to the current procedure. For instance

$$n = n_{\mathfrak{0}}$$

says that the variable n has not changed since entry to the current procedure. Thus our $\mathfrak{0}$ subscript is similar to the `old` operator found in JML [LBR06] and Spec# [BLS05]).

Formally, we call such subscripts *time indices* and define the set of them to be

$$\mathit{Time} := \{\mathfrak{0}, \mathfrak{I}, \mathfrak{2}, \mathfrak{3}, \mathfrak{C}\}$$

We call $\mathfrak{C}$ the "current time", $\mathfrak{I}$ the "previous time" and $\mathfrak{0}$ the "starting time". Figure 3.6 shows how this scheme works for an intraprocedural statement. As a shorthand, unadorned variables will refer to the current time $\mathfrak{C}$. We will use variables $\mathfrak{j}$ and $\mathfrak{k}$ to range over time indices. (For procedure calls and returns, two further indices $\mathfrak{2}$ and $\mathfrak{3}$ are needed; this will be described in Subsection 4.4.1.)

3.2.2 Syntax and semantics of our logic $\mathscr{L}$

We now give syntax and semantics for our logic $\mathscr{L}$; we have chosen to make $\mathscr{L}$ a first order logic with transitive closure, or FO(TC). This important design decision will be discussed later (Section 3.7) The syntax of the logic is given in Figure 3.7.

$$
\begin{aligned}
\text{term} ::= {} & v_{\mathfrak{j}} \quad \text{(program variable)} \\
| {} & X \quad \text{(logical variable)} \\
| {} & n \quad \text{(integer constant)} \\
| {} & \text{term} \otimes \text{term} \quad \text{(arithmetic)} \\
| {} & f_{\mathfrak{j}}\ (\ \text{term}\ ) \quad \text{(field lookup)}
\end{aligned}
$$

$$
\text{literal} ::= \text{term} = \text{term} \mid \text{term} < \text{term} \mid \text{term} \leq \text{term} \mid \mathit{allocd}_{\mathfrak{j}}(\ \text{term}\ )
$$

$$
\begin{aligned}
\Phi ::= {} & \text{literal} \mid \mathit{True} \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \rightarrow \Phi \mid \Phi \leftrightarrow \Phi \\
| {} & \exists X\ \Phi \mid \forall X\ \Phi \mid \mathit{TC}_{[A,B]}\,[\Phi(A,B)]\,(\mathit{term},\mathit{term})
\end{aligned}
$$

Figure 3.7: Grammar of the logic $\mathscr{L}$ which describes program states. $\mathscr{L}$ is a first order logic with a transitive closure operator TC. Here $\mathfrak{j} \in \mathit{Time}$, $v \in \mathit{Vars}$ and $f \in \mathit{Fields}$. X, A and B are logical variables (i.e. variables not appearing in the program) which range over integer values.

$$
I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t) = \begin{cases}
\mathit{env}_{\mathfrak{j}}(v) & \text{if } t \text{ is } v_{\mathfrak{j}} \text{ (program variable)} \\
\rho(X) & \text{if } t \text{ is } X \text{ (logical variable)} \\
n & \text{if } t \text{ is } n \text{ (integer constant)} \\
\begin{array}{c} I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t') \\ \otimes I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t'') \end{array} & \text{if } t \text{ is } t' \otimes t'' \text{ (arithmetic)} \\
h_{\mathfrak{j}}(f, I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t')) & \text{if } t \text{ is } f_{\mathfrak{j}}(t') \text{ (field lookup)}
\end{cases}
$$

where each $s_{\mathfrak{j}}$ is $(\mathit{env}_{\mathfrak{j}}, h_{\mathfrak{j}}, A_{\mathfrak{j}})$

Figure 3.8: Interpretation of terms in our logic $\mathscr{L}$.

Semantics of formulae: $(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \in$:

$$
\begin{array}{lll}
[\![t = t']\!]_\rho & \text{iff} & I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t) = I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t') \\
[\![t < t']\!]_\rho & \text{iff} & I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t) < I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t') \\
[\![t \leq t']\!]_\rho & \text{iff} & I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t) \leq I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t') \\
[\![\mathit{allocd}_{\mathfrak{j}}(t)]\!]_\rho & \text{iff} & I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t) \in A_{\mathfrak{j}} \\
[\![\neg\Phi]\!]_\rho & \text{iff} & (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \notin [\![\Phi]\!]_\rho \\
[\![\Phi \wedge \Phi']\!]_\rho & \text{iff} & (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \in [\![\Phi]\!]_\rho \cap [\![\Phi']\!]_\rho \\
[\![\Phi \vee \Phi']\!]_\rho & \text{iff} & (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \in [\![\Phi]\!]_\rho \cup [\![\Phi']\!]_\rho \\
[\![\Phi \rightarrow \Phi']\!]_\rho & \text{iff} & (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \notin [\![\Phi]\!]_\rho \text{ or } (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \in [\![\Phi']\!]_\rho \\
[\![\Phi \leftrightarrow \Phi']\!]_\rho & \text{iff} & (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \in [\![\Phi]\!]_\rho \cap [\![\Phi']\!]_\rho \\
 & & \text{or } (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \notin [\![\Phi]\!]_\rho \cup [\![\Phi']\!]_\rho \\
[\![\exists X \Phi]\!]_\rho & \text{iff} & \exists n \in \mathbb{Z} \text{ s.t. } (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \in [\![\Phi]\!]_{\rho \oplus \{X \mapsto n\}} \\
[\![\forall X \Phi]\!]_\rho & \text{iff} & \forall n \in \mathbb{Z}, (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \in [\![\Phi]\!]_{\rho \oplus \{X \mapsto n\}} \\
[\![TC_{[A,B]} \left[\Phi(A,B)\right](t,t')]\!]_\rho & \text{iff} & \begin{array}{l} \text{for some } n^1, \ldots, n^k \in \mathbb{Z} \text{ we have:} \\ I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t) = n^1, \\ I(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}, \rho, t') = n^k \\ \text{and for } j = 1 \ldots k-1, \\ \quad (s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \in [\![\Phi]\!]_{\rho \oplus \{A \mapsto n^j, B \mapsto n^{j+1}\}} \end{array}
\end{array}
$$

where each $s_{\mathfrak{j}}$ is $(\mathit{env}_{\mathfrak{j}}, h_{\mathfrak{j}}, A_{\mathfrak{j}})$

Figure 3.9: Semantics of the logic $\mathscr{L}$ which describes program states. Formulae of $\mathscr{L}$ are interpreted over five states, which are intended as "snapshots" of the program's execution, taken at different times. Hence, formulae can describe the effects of atomic statements, and frame conditions. The operator $\oplus$ denotes function override, and the term interpretation function I is from Figure 3.8.

Variables, fields and the *allocd*(x) predicate, which expresses that memory address x has been allocated to a heap object, can all be decorated with a time index $\mathfrak{j} \in \mathit{Time}$.

Quantification is allowed only over (integer-valued) *logical variables* which are kept separate from program variables and capitalised. Recall that, informally, the transitive closure operator TC works as follows: $TC_{[A,B]}\left[\Phi(A,B)\right](t,t')$ says that from t we can "reach" t' via some path of intermediate points, such that for each pair of successive points (A,B) along the path we have $\Phi(A,B)$.

Figures 3.8 and 3.9 define the logic's semantics: to account for the time indices, formulae are interpreted in a 5-tuple of states, with the five states corresponding to the five time indices. Figure 3.8 shows how to interpret terms to their values, and then Figure 3.9 defines $[\![-]\!]_\rho$ as a subset of State^5. The expression $(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}}) \in [\![\Phi]\!]_\rho$ means that Φ is true of the 5-tuple of states $(s_{\mathfrak{0}}, s_{\mathfrak{1}}, s_{\mathfrak{2}}, s_{\mathfrak{3}}, s_{\mathfrak{C}})$ where ρ gives values to the logical variables. When Φ contains no free variables we omit ρ.

Sometimes we will be handling formulae containing only current variables, fields and allocation predicates. We call this sublogic $\mathscr{L}^{\{\mathfrak{C}\}}$. In this case the truth value of the formula depends only on the current state, and we use a unary semantics $[\![-]\!]^{\{\mathfrak{C}\}} \subseteq \mathit{State}$. More generally, for any subset $T \subseteq \mathit{Time}$ of cardinality k, we can easily get a k-ary semantics $[\![-]\!]^T \subseteq \mathit{State}^k$ for formulae whose time indices come only from T; we call the sublogic of such formulae $\mathscr{L}^T$.

Finally we have notation for "time substitution": by $\Phi[\mathfrak{j}\backslash\mathfrak{k}]$ (where $\mathfrak{j},\mathfrak{k} \in \mathit{Time}$) we denote the result of substituting in Φ the time index $\mathfrak{k}$ for each occurrence of the time index $\mathfrak{j}$.

3.3 Abstract models of programs

3.3.1 Syntactic definition

We now introduce, syntactically, the kind of models we will build from input programs.

From now on, we are going to assume that all our domains have a power set structure. Working with power sets simplifies things very much: it allows us to work at the level of individual elements rather than at the level of sets of elements:

> "When the lattice is the power set of some basic finite set D [...], the algorithm can be modified to propagate elements of D instead of elements of 2^D." [RRL99]

Bear in mind that our task here is to perform *initial exploration* of how to build a verification system that combines analysis modules, so we want to keep things simple initially. The process of extracting counterexample paths is also simplified, as is that of handling procedures:

> "By restricting domains to be power sets [...], we are able to efficiently create simple representations of functions that summarize the effects of procedures (by supporting efficient lookup operations from input facts to output facts)." [RHS95]

Our choice to stick to power set domains is implicit in the following definition. This decision *doesn't* mean we *can't use* other domains, but it may mean we lose some of their useful structure by treating them as power set domains; this will be the case with our three-valued shape graph domain.

Definition 3.3.1. Given a program $P = ([f_1, \ldots, f_m], [\Pi_1, \ldots, \Pi_n])$, an *abstract model of* P is a tuple $Q = (T, \gamma, N, \mathit{Edges}, \mathit{CallEdges}, \mathit{ReturnEdges})$ where:

- T is a set of *abstract values*
- $\gamma : T \to \mathbb{P}(State \times State)$ is a concretisation function
- N is the set of nodes of the abstract model, where each node is a triple (π, l, a) such that: π names a procedure Π in P, $l \in Nodes(Graph(\Pi))$ i.e. l is a control location in procedure Π, and $a \in T$ i.e. a is an abstract value
- *Edges*, *CallEdges*, *ReturnEdges* are transition relations i.e. subsets of $N \times N$, whose elements we refer to as *ordinary edges*, *call edges* and *return edges* respectively □

In contrast to Chapter 2, our abstract values now abstract *pairs of states* rather than single states. This is because we want to track frame conditions; for example we might want $n = n_{\mathbf{0}}$ to be an abstraction predicate. The first state of the pair is the state at procedure entry, and the second is the current state.

Running example:

By way of example, we show in Figure 3.10 a model for our earlier program, built using a sign analysis. Here the abstract values (that is, elements of T) are finite partial maps from variables to signs:

$$T_{sign} := Vars \rightharpoonup_{\text{fin}} \{\text{pos}, \text{neg}, \text{zero}\}$$

The entries in the map are read conjunctively, so that each variable must have the sign indicated. Also, the signs refer to the *current* values of the variables; the *starting* state (that on entry to the current procedure) is left unconstrained. Formally we define first a helper function $\hat{\gamma}$:

$$\begin{aligned} \hat{\gamma}(v, \text{pos}) &:= \{(e, h, A) \mid e(v) > 0\} \\ \hat{\gamma}(v, \text{neg}) &:= \{(e, h, A) \mid e(v) < 0\} \\ \hat{\gamma}(v, \text{zero}) &:= \{(e, h, A) \mid e(v) = 0\} \end{aligned}$$

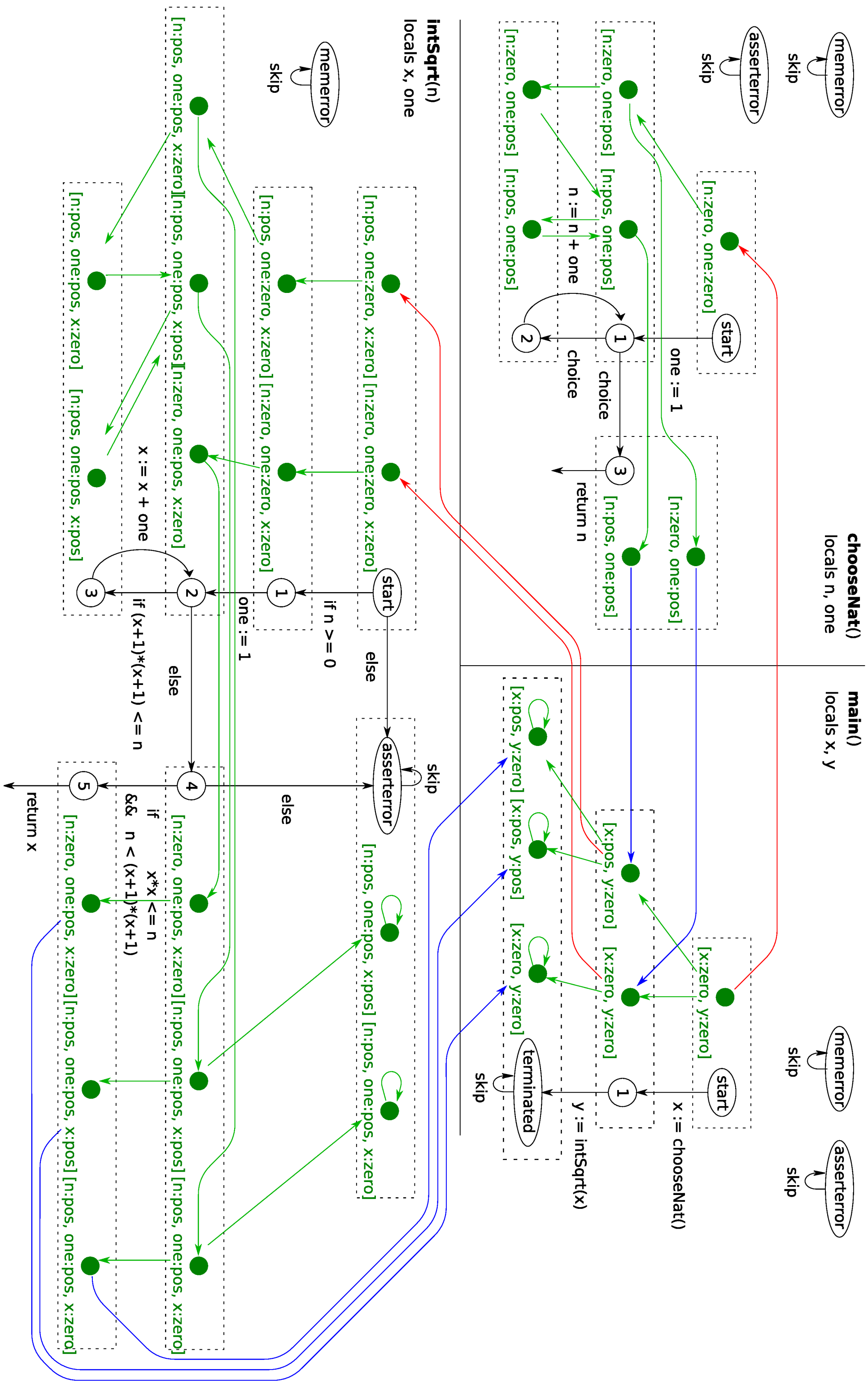

Figure 3.10: An example of an abstract model, built from a sign analysis. The relations *Edges*, *CallEdges* and *ReturnEdges* are shown in green, red and blue respectively. (This graph is hand-edited for better layout, though our implementation can generate similar graphs; see Chapter 4.)

and then the concretisation function γ:

$$\gamma_{sign}(M) \quad := \quad State \times \bigcap_{v \in dom(M)} \hat{\gamma}(v, M(v))$$

Here if the domain of M is empty, the intersection is interpreted as all of *State*.

3.3.2 Sound abstract models

Of course, as stated back in Chapter 1, we can only use abstract models for verification if there is an appropriate relationship between the abstract model and the original program. In the development of Chapter 2 this relationship was provided by the conditions in Remark 2.3.5; another such relationship is simulation (e.g. [BG03]).

We now define what it means for an abstract model to be a *sound* model of the original program. Our definition is non-standard in that it ensures that procedure calls and returns are correctly treated, and accounts for the need to be able to refer to the program's state on entry to the current procedure (i.e. at the "starting" time).

Definition 3.3.2. Given a program P and an abstract model Q of P as above, we say that Q is *sound* if the following conditions are satisfied:

sound-init There exists $a \in T$ such that $(\pi_1, start, a) \in N$ and $(s^{start}, s^{start}) \in \gamma(a)$.

sound-intra If $\pi, s_0 \xrightarrow{l,s} l', s'$ with $(\pi, l, a) \in N$ and $(s_0, s) \in \gamma(a)$, then there exists $a' \in T$ such that $(\pi, l', a') \in N$, $(s_0, s') \in \gamma(a')$ and $((\pi, l, a), (\pi, l', a')) \in Edges$.

sound-call If $\pi, s_0 \rightarrow l, s : \pi', s'$ with $(\pi, l, a) \in N$ and $(s_0, s) \in \gamma(a)$ then there exists $a' \in T$ such that $(\pi', start, a') \in N$, $(s', s') \in \gamma(a')$ and also $((\pi, l, a), (\pi', start, a')) \in CallEdges$.

sound-return If $\pi, s_0 \xrightarrow{l_1,s_1,\bar{\pi},s_2,l_3,s_3} l', s'$ and N has elements (π, l_1, a_1), $(\bar{\pi}, \mathit{start}, a_2)$ and $(\bar{\pi}, l_3, a_3)$ such that

$(s_0, s_1) \in \gamma(a_1)$

$(s_2, s_3) \in \gamma(a_3)$

$((\pi, l_1, a_1), (\bar{\pi}, \mathit{start}, a_2)) \in \mathit{CallEdges}$

$((\bar{\pi}, \mathit{start}, a_2), (\bar{\pi}, l_3, a_3)) \in \mathit{Edges}^*$

then N also contains an element (π, l', a') such that

$(s_0, s') \in \gamma(a')$

$((\bar{\pi}, l_3, a), (\pi, l', a')) \in \mathit{ReturnEdges}$

$((\pi, l_1, a_1), (\pi, l', a')) \in \mathit{Edges}$

(Here and henceforth, R^* denotes the reflexive transitive closure of the binary relation R.) □

Next we prove a convenient property of sound abstract models; this is a stepping stone which will allow us to justify our use of sound models for verification.

Theorem 3.3.3. A consequence of soundness Given a program P and a sound abstract model Q of P as above, the following condition holds:

sound-coverage If $\pi, s_0 \xrightarrow{p} l, s$ then:

1. N contains at least one element $(\pi, \mathit{start}, a_0)$ such that $(s_0, s_0) \in \gamma(a_0)$, and
2. for each such element, N contains an element (π, l, a) such that $(s_0, s) \in \gamma(a)$ and $((\pi, \mathit{start}, a_0), (\pi, l, a)) \in \mathit{Edges}^*$.

(The subtlety here is that in general distinct abstract states need not have disjoint concretisations, and thus the concrete starting state s_0 might lie within the concretisation of more than one abstract node; from *all* such abstract nodes there must be a path to a node representing s.)

Proof: This property talks about paths of edges through the model, so it is not surprising that the proof is by induction. We use the following condition:

ind-coverage(k) If $\pi, s_0 \xrightarrow{p} l, s$ can be derived in k steps or fewer, then:

1. N contains at least one element $(\pi, \mathit{start}, a_0)$ such that $(s_0, s_0) \in \gamma(a_0)$, and
2. for each such element, N contains an element (π, l, a) such that $(s_0, s) \in \gamma(a)$ and $((\pi, \mathit{start}, a_0), (\pi, l, a)) \in \mathit{Edges}^*$.

The base case $k = 0$ is trivial: there are no derivations of zero steps. Now we do the inductive case: assume ind-coverage(k) and prove ind-coverage$(k + 1)$. So let $\pi, s_0 \xrightarrow{p} l', s'$ be derivable in $k + 1$ steps or fewer. We split our analysis into cases depending on which rule was used to complete the derivation: there are four subcases depending on the form of p.

- $p = l, s$: Here one of the intraprocedural rules was used to complete the derivation, and from its premises we have $\pi, s_0 \xrightarrow{q} l, s$ derivable in k steps or fewer.

  Applying part 1. of ind-coverage(k) to this, there exists an element $(\pi, \mathit{start}, a_0)$ of N such that $(s_0, s_0) \in \gamma(a_0)$; this meets part 1. of ind-coverage$(k + 1)$.

  For any such element, part 2. of ind-coverage(k) gives us another element (π, l, a) of N such that $(s_0, s) \in \gamma(a)$ and $((\pi, \mathit{start}, a_0), (\pi, l, a)) \in \mathit{Edges}^*$.

  Applying sound-intra to $\pi, s_0 \xrightarrow{l,s} l', s'$, we see that N contains an element (π, l', a') such that $(s_0, s') \in \gamma(a')$ and $((\pi, l, a), (\pi, l', a')) \in \mathit{Edges}$, whence $((\pi, \mathit{start}, a_0), (\pi, l', a')) \in \mathit{Edges}^*$ and part 2. of ind-coverage$(k + 1)$ is met.

- $p = \epsilon$ and $\pi = \pi_1$: Here the init rule was used to complete the derivation, and the conclusion has the form $\pi_1, s^{start} \xrightarrow{\epsilon} start, s^{start}$ i.e. $l' = start$ and $s' = s_0 = s^{start}$.

  Using sound-init, there exists $a_0 \in T$ such that $(\pi_1, start, a_0) \in N$ and $(s^{start}, s^{start}) \in \gamma(a_0)$, i.e. $(s_0, s_0) \in \gamma(a_0)$, meeting part 1. of ind-coverage(k+1).

  For each such element, putting $a' = a_0$ we see that N contains an element (π, l', a') such that $(s_0, s') \in \gamma(a')$. Also $((\pi, start, a_0), (\pi, l', a')) \in Edges^*$ simplifies to $((\pi, start, a_0), (\pi, start, a_0)) \in Edges^*$ which is trivially true.

- $p = \epsilon$ and $\pi \neq \pi_1$: Here the call-2 rule was used to complete the derivation, and the conclusion is of the form $\pi, s' \xrightarrow{\epsilon} start, s'$, i.e. we have $l' = start$ and $s_0 = s'$.

  From the premise of call-2 we also have that $\hat{\pi}, \hat{s}_0 \rightarrow \hat{l}, \hat{s} : \pi, s'$ (using hats for the caller procedure), and this must be derivable in k steps or fewer. This can only be obtained using the call-1 rule, so from the premises of call-1 we have $\hat{\pi}, \hat{s}_0 \xrightarrow{\hat{p}} \hat{l}, \hat{s}$ which must be derivable in $k - 1$ or fewer steps.

  Applying ind-coverage(k) to this, we see that N contains an element $(\hat{\pi}, \hat{l}, a)$ such that $(\hat{s}_0, \hat{s}) \in \gamma(a)$.

  Applying sound-call to this, we see that N also contains an element $(\pi, start, a_0)$ such that $(s', s') \in \gamma(a_0)$; this meets part 1. of ind-coverage($k + 1$).

  For each such element, putting $a' = a_0$ we see that N contains an element (π, l', a') such that $(s_0, s') \in \gamma(a')$. Also $((\pi, start, a_0), (\pi, l', a')) \in Edges^*$ simplifies to $((\pi, start, a_0), (\pi, start, a_0)) \in Edges^*$ which is trivially true.

- $p = l_1, s_1, \bar{\pi}, s_2, l_3, s_3$: Here the return rule was used to complete the derivation, and the conclusion is of the form

$$\pi, s_0 \xrightarrow{l_1, s_1, \bar{\pi}, s_2, l_3, s_3} l', s'$$

  From the premises of the return rule, we have $\bar{\pi}, s_2 \xrightarrow{\bar{p}} l_3, s_3$ and $\pi, s_0 \rightarrow l_1, s_1$:

$\bar{\pi}, s_2$ which must each be derivable in $k-1$ or fewer steps. The only way to obtain the latter is by call-1, and therefore $\pi, s_0 \xrightarrow{q} l_1, s_1$ is derivable in $k-2$ or fewer steps.

Applying ind-coverage(k) part 1. to $\pi, s_0 \xrightarrow{q} l_1, s_1$, we see that N contains an elements $(\pi, \mathit{start}, a_0)$ such that $(s_0, s_0) \in \gamma(a_0)$, meeting part 1. of ind-coverage($k+1$).

To show ind-coverage($k+1$) part 2., consider *any* $(\pi, \mathit{start}, a_0) \in N$ such that $(s_0, s_0) \in \gamma(a_0)$. By ind-coverage($k$) part 2., there exists $(\pi, l_1, a_1) \in N$ such that $(s_0, s_1) \in \gamma(a_1)$ and $(\pi, \mathit{start}, a_0), (\pi, l_1, a_1)) \in \mathit{Edges}^*$.

Applying sound-call, we see that N also contains an element $(\bar{\pi}, \mathit{start}, a_2)$ such that $(s_2, s_2) \in \gamma(a_2)$ and $((\pi, l_1, a_1), (\bar{\pi}, \mathit{start}, a_2)) \in \mathit{CallEdges}$.

This is where the quantification implicit in part 2. of ind-coverage is necessary to make the proof go through. Applying ind-coverage(k) to $\bar{\pi}, s_2 \xrightarrow{\bar{p}} l_3, s_3$, we find that N contains an element $(\bar{\pi}, l_3, a_3)$ such that $(s_2, s_3) \in \gamma(a_3)$ and $((\bar{\pi}, \mathit{start}, a_2), (\bar{\pi}, l_3, a_3)) \in \mathit{Edges}^*$.

We have now assembled everything needed to invoke sound-return, which tells us that N also contains an element (π, l', a') such that $(s_0, s') \in \gamma(a')$ and $((\pi, l_1, a_1), (\pi, l', a')) \in \mathit{Edges}$. To finish we put together $((\pi, l_1, a_1), (\pi, l', a')) \in \mathit{Edges}$ and $(\pi, \mathit{start}, a_0), (\pi, l_1, a_1)) \in \mathit{Edges}^*$ and conclude that $(\pi, \mathit{start}, a_0), (\pi, l', a')) \in \mathit{Edges}^*$. This meets part 2. of ind-coverage($k+1$).

Thus, ind-coverage(k) holds for all k. To finish, simply note that because all derivations are finite, sound-coverage follows. □

3.3.3 Using sound models to verify programs

As we explained in Subsection 2.1.2, we will transform our verification problems into reachability problems. Hence, given a set of "bad" locations $B \subseteq \mathit{ProcNames} \times \mathit{Locs}$,

we will then want to determine whether any of the bad locations are reachable, i.e. whether

$$\exists(\pi, l) \in B \;\; \exists s_0, s, p \quad \pi, s_0 \xrightarrow{p} l, s$$

The following Criterion shows how to use a sound model to falsify such a condition (and thus verify the program).

Criterion 3.3.4. Given a program P and a sound abstract model Q of P as above, and a set of bad nodes B, then

$$(1.) \qquad \text{there do not exist } a \in T, (\pi, l) \in B \text{ such that } (\pi, l, a) \in N$$

is sufficient to falsify

$$(2.) \qquad \exists(\pi, l) \in B \;\; \exists s_0, s, p \quad \pi, s_0 \xrightarrow{p} l, s$$

i.e. is sufficient to show that no bad nodes are reached.

Proof: We shall assume (2.) and prove the negation of (1.). From (2.) we see that there exists $(\pi, l) \in B$ and s_0, s, p such that $\pi, s_0 \xrightarrow{p} l, s$. By the soundness of Q and using Theorem 3.3.3, there exists $a \in T$ such that $(\pi, l, a) \in N$ and $(s_0, s) \in \gamma(a)$. □

Typically we will be interested in the locations used to model memory errors and assertion violations, i.e. we will take $B := \{\pi_1, \ldots, \pi_n\} \times \{\mathit{memerror}, \mathit{asserterror}\}$.

3.4 Analysis modules

In this subsection we present and discuss the notion of an *analysis module* which is central to our work. Intuitively an analysis module is a program analysis tool which has been appropriately wrapped for integration into our system.

3.4.1 Our interface for analysis modules

Definition 3.4.1. The *Stmt* type. The type *Stmt* has elements of the following forms:

- `Skip`
- `VarCopy`(u, v)
- `AssignConst`(u, k)
- `Arith`$(u, v_1, \otimes, v_2)$
- `FieldRead`(u, v, f)
- `FieldWrite`(v, f, u)
- `New`(u, v) □

Values of *Stmt* are like the edge labels in CFGs, except that (for reasons explained shortly) there are no cases for `If`, `Choice`, `Call` or `Return`.

Definition 3.4.2. An *analysis module* **M** is a software module implementing the following interface:

- **finite datatype** T
- **function** $share : T \times Stmt \times ProcNames \to \mathscr{L}^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$
- **function** $succ : T \times Stmt \times ProcNames \times \mathscr{L}^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}} \to \mathbb{P}(T)$
- **function** $succ_C : T \times ProcNames \times ProcNames \times Vars\ \text{list} \to \mathbb{P}(T)$
- **function** $succ_R : T \times T \times ProcNames \times ProcNames \times Vars \times Vars \times Vars\ \text{list} \to \mathbb{P}(T)$
- **function** $init : \{\cdot\} \to \mathbb{P}(T)$

and equipped with a notional (i.e. not actually implemented) concretisation function

- **function** $\gamma : T \to \mathbb{P}(State \times State)$

(Here and henceforth, the type "X list" is inhabited by finite lists of values of type X. The 'list' type constructor has higher binding power than does the product operator $\times$.) □

(The `If` and `Choice` cases are missing from *Stmt* because, as we will see, these are translated into suitable invocations of *succ* with `Skip`; similarly `Call` and `Return` are missing because they are handled by the $succ_C$ and $succ_R$ functions instead.)

When working with several modules, we prevent confusion by referring to **M** 's components as **M**.*share*, **M**.*succ* and so on. The role of each interface component is as follows.

- The datatype T is the type of the abstract values used by the module.
- The (notional) concretisation function γ gives meaning to the abstract values (elements of T), as it did in Subsection 2.3.2.

  The only difference is that, as mentioned previously, our abstract values now abstract *pairs of states* rather than single states, to allow for frame conditions etc.. The first state of the pair is the state at procedure entry, and the second is the current state.
- Calling $share(a, s, \pi)$ asks the module to share an $\mathscr{L}$-formula Φ which is valid in all concrete state pairs represented by the abstract state a (i.e. Φ is entailed by a) and might be useful to other modules when computing successors for the statement s in procedure π.
- Calling $succ(a, s, \pi, \Phi)$ computes the set of abstract states the program may reach by executing the statement s in a concrete state represented by a and

satisfying the formula Φ. In practice Φ will be the information gathered from the other modules by *share*. (Thus *succ* is similar to the abstract transfer functions $f^{\#}$ of Subsection 2.3.2 but with the formula Φ as an extra argument.)

- The function $succ_C$ generates successors for call statements, as *succ* does for ordinary statements. In the invocation $succ_C(a, \pi, \pi', [a_1, \ldots, a_k])$, a is the abstract value at the call point, π names the calling procedure, π' names the called procedure and $[a_1, \ldots, a_k]$ are the actual parameters.

- Returning from a procedure call is treated by a similar function $succ_R$, except that two abstract values must be supplied instead of one: one describing the callee's state at the return point, and one describing the caller's state when the call was made. Roughly, constraints on the heap after the return are taken from the first, whereas constraints on the caller's local variables are taken from the second.

  In the invocation $succ_R(a, a', \pi, \pi', x, r, [a_1, \ldots, a_k])$, π and π' name respectively the caller and called procedures, $[a_1, \ldots, a_k]$ are the call's actual parameters, a is the abstract value at the point where the call was made, a' is the abstract value at the return point, r is the variable returned by the called procedure, and x is the variable in the caller waiting to receive the result.

- *init* is used to start off the analysis, giving a set of abstract values to represent the initial state of the program.

Note that modules also have access to the whole program that is being analysed; this can be thought of as an implicit parameter to all the functions outlined above.

3.4.2 An example analysis module: multi-variable sign analysis

We now construct in detail an example analysis module, implementing a multi-variable sign analysis. This analysis module can be used to automatically construct the model in Figure 3.10 (page 104). We will not go into the same level of detail for our other analysis modules, but wish to include a fully specified example module, and give the reader an idea of what is involved in writing such a module.

Like most of the analyses we will deal with, the multi-variable sign analysis is "configurable" or "tunable": we can choose, for each procedure in our target program, which variables we would like to track signs for, and which we would not. Thus formally instead of defining an analysis module **compsigns** , we define an *indexed family* of analysis modules, **compsigns** $\langle\Delta\rangle$, parametrised by this choice, which we represent as a mapping

$$\Delta : \mathit{Procs}(P) \to \mathit{Vars}\ \text{list}$$

Here the abstract values (elements of T) are finite partial maps from variables to signs:

$$T \quad := \quad \{M \in \mathit{Vars} \rightharpoonup_{\text{fin}} \{\text{pos}, \text{neg}, \text{zero}\} \mid \mathit{dom}(M) = \Delta(\pi) \text{ for some } \pi \in \mathit{Procs}(P)\}$$

The concretisation function γ is the same as on page 105. The definition of the information-sharing function *share* is simple (again using a helper function):

$$\begin{aligned} \mathit{sharehelper}(v, \text{pos}) &:= v_{\mathbf{I}} > 0 \\ \mathit{sharehelper}(v, \text{neg}) &:= v_{\mathbf{I}} < 0 \\ \mathit{sharehelper}(v, \text{zero}) &:= v_{\mathbf{I}} = 0 \end{aligned}$$

$$share(M, s, \pi) \quad := \quad \bigwedge_{v \in dom(M)} sharehelper(v, M(v))$$

For this sign analysis, the shared formula entirely captures the abstract value $M \in T$ but this will not always be possible or desirable: generally any formula entailed by the abstract value is enough. The only subtlety is that each variable v is encoded with the corresponding "previous time" variable $v_{\mathfrak{l}}$, to indicate that we are describing the state before execution of the upcoming statement.

To implement the successor function *succ*, we need a way to reason about how the possible signs of variables are affected by the various forms of statements. We do this in a *compositional* way: for each arithmetic operator

$$\otimes : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

we provide, overloading the symbol $\otimes$, an abstract operator

$$\otimes : \{\text{pos}, \text{neg}, \text{zero}, \text{any}\} \times \{\text{pos}, \text{neg}, \text{zero}, \text{any}\} \to \{\text{pos}, \text{neg}, \text{zero}, \text{any}\}$$

Tables for these operators are found in Appendix A.2. The sign of an expression is then safely approximated by "lifting" $M \in T$ from *Vars* to whole terms, as follows:

$$\begin{aligned}
M(v_{\mathfrak{j}}) &= \text{if } \mathfrak{j} \neq 1 \text{ or } v \notin dom(M) \text{ then any else } M(v) \\
M(X) &= \text{any} \\
M(n) &= \text{if } n > 0 \text{ then pos else if } n = 0 \text{ then zero else neg} \\
M(t \otimes t') &= M(t) \otimes M(t') \\
M(f_{\mathfrak{j}}(t)) &= \text{any}
\end{aligned}$$

As can be seen, we make no attempt to model heap fields, and treat the result of all heap reads as 'any'. The same is done for logical variables. For convenience we

also overload the symbol $\oplus$, defining a helper function

$$\oplus : T \times \mathit{Vars} \times \{\text{pos}, \text{neg}, \text{zero}, \text{any}\} \to \mathbb{P}(T)$$

which, roughly, abstracts function overriding:

$$\oplus(M, v, \sigma) = \begin{cases} \{M\} & \text{if } v \notin \mathit{dom}(M) \\ \{M \oplus \{v \mapsto \sigma\}\} & \text{if } v \in \mathit{dom}(M) \text{ and } \sigma \neq \text{any} \\ \left\{ \begin{array}{l} M \oplus \{v \mapsto \text{pos}\}, \\ M \oplus \{v \mapsto \text{zero}\}, \\ M \oplus \{v \mapsto \text{neg}\} \end{array} \right\} & \text{if } v \in \mathit{dom}(M) \text{ and } \sigma = \text{any} \end{cases}$$

We are now in a position to have a first go at defining the successor function:

$$\begin{aligned} \mathit{succ}'(M, \texttt{Skip}, \pi, \Phi) &= \{M\} \\ \mathit{succ}'(M, \texttt{VarCopy}(u, v), \pi, \Phi) &= \oplus(M, u, M(v)) \\ \mathit{succ}'(M, \texttt{AssignConst}(u, k), \pi, \Phi) &= \oplus(M, u, M(k)) \\ \mathit{succ}'(M, \texttt{Arith}(u, v_1, \otimes, v_2), \pi, \Phi) &= \oplus(M, u, M(v_1 \otimes v_2)) \\ \mathit{succ}'(M, \texttt{FieldRead}(u, v, f), \pi, \Phi) &= \oplus(M, u, \text{any}) \\ \mathit{succ}'(M, \texttt{FieldWrite}(v, f, u), \pi, \Phi) &= \{M\} \\ \mathit{succ}'(M, \texttt{New}(u, v), \pi, \Phi) &= \oplus(M, u, \text{pos}) \end{aligned}$$

When the new sign of a variable cannot be determined i.e. evaluates to 'any', the module branches, producing three new states corresponding to the three possible signs pos, neg and zero of the variable. This branching occurs in the definition of the abstract $\oplus$.

This successor function is fine as far as it goes, and soundly treats statements. What *succ*′ doesn't do, however, is make any use of the parameter Φ, that is, the extra information provided by other analysis modules or by guards from loops or

conditionals in the program. In particular, if we see that Φ contradicts the sign information held in M, we would like the module to return an empty set of successors. To enable this, we define a function *check* such that $check(\Phi, M)$ evaluates a formula Φ according to the sign information in M. Then *succ* is defined as follows:

$$succ(M, s, \pi, \Phi) \quad := \quad \begin{cases} \emptyset & \text{if } check(\Phi, M) = false \\ succ'(M, s, \pi, \Phi) & \text{otherwise} \end{cases}$$

To obtain *check*, we give a three-valued Kleene-style semantics to formulae of $\mathscr{L}$, taking an $M \in T$ as our model. (Appendix A.1 gives an introduction to three-valued logic for readers unfamiliar with it.) This approach accounts very naturally for the fact that M contains only limited information about the program state, and allows us to make the most of the information we do have: even if a formula Φ contains a subformula on which M can shed no light, we may still be able to obtain a definite truth value for Φ.

In fact, the use of a three-valued semantics to soundly evaluate first-order formulae with respect to a set of sign constraints is a novelty of our work, as far as we are aware. We regard it as a natural step, however, since three-valued semantics have been used for other aspects of program analysis.

To implement *check* we must supply three-valued interpretations of all the constructors used to make $\mathscr{L}$-formulae. Then to find $check(\Phi, M)$ we substitute $M(t)$ for each term t in Φ, and evaluate the resulting expression under the three-valued semantics. For the propositional connectives, the usual Kleene semantics, given in Appendix A.1, is used. Abstract versions of the atomic predicates $=$, $<$, $\leq$ and $allocd_{\mathfrak{f}}$ are given by tables in Appendix A.2. The quantifiers $\exists X$ and $\forall X$ are simply interpreted as the identify function; this is sound because logical variables are mapped to 'any' by each (lifted) $M \in T$. *TC* subformulae are treated completely imprecisely, that is, always evaluated as *unknown*.

Consider for example

$$\mathit{check}(\quad \forall X(v \times X > 0 \rightarrow \mathit{allocd}_{\mathbf{I}}(X)),\quad \{v \mapsto \text{zero}\}\quad)$$

We begin by substituting as described above, obtaining

$$\forall X(\quad \{v \mapsto \text{zero}\}(v \times X) > \{v \mapsto \text{zero}\}(0)\quad \rightarrow\quad \mathit{allocd}_{\mathbf{I}}(\{v \mapsto \text{zero}\}(X))\quad)$$

and then evaluating as follows:

$$\begin{aligned}
&\rightsquigarrow \quad \{v \mapsto \text{zero}\}(v \times X) > \{v \mapsto \text{zero}\}(0) \quad \rightarrow \quad \mathit{allocd}_{\mathbf{I}}(\{v \mapsto \text{zero}\}(X))\\
&\rightsquigarrow \quad \{v \mapsto \text{zero}\}(v \times X) > \{v \mapsto \text{zero}\}(0) \quad \rightarrow \quad \mathit{allocd}_{\mathbf{I}}(\text{any})\\
&\rightsquigarrow \quad \{v \mapsto \text{zero}\}(v \times X) > \{v \mapsto \text{zero}\}(0) \quad \rightarrow \quad \text{unknown}\\
&\rightsquigarrow \quad \{v \mapsto \text{zero}\}(v) \times \{v \mapsto \text{zero}\}(X) > \text{zero} \quad \rightarrow \quad \text{unknown}\\
&\rightsquigarrow \quad \text{zero} \times \text{any} > \text{zero} \quad \rightarrow \quad \text{unknown}\\
&\rightsquigarrow \quad \text{zero} > \text{zero} \quad \rightarrow \quad \text{unknown}\\
&\rightsquigarrow \quad \text{false} \quad \rightarrow \quad \text{unknown}\\
&\rightsquigarrow \quad \text{true}
\end{aligned}$$

To complete the analysis module it remains to define the initialisation function *init* which begins execution, and the procedure call and return functions succ_C and succ_R. The definition of *init* simply reflects the fact that all program variables in the main procedure π_1 start off with value zero (recall that local variables are set to zero on procedure entry, and the main procedure is parameterless):

$$\mathit{init}(\cdot) \quad := \quad \bigcup_{v \in \Delta(\pi_1)} \{v \mapsto \text{zero}\}$$

For procedure calls from procedure π to procedure π', local variables again begin as zero. To obtain the sign of each formal parameter, we look up the actual parameter

a_i in the abstract constraint at the call site. When this is not possible, because the actual parameter's sign is not tracked in the calling procedure π (i.e. is not in $\Delta(\pi)$) we must branch for all possibilities. Thus we define $succ_C(M, \pi, \pi', [a_1, \ldots, a_k])$ to be the set of those $M' \in T$ such that:

1. $dom(M') = \Delta(\pi')$
2. For each local variable, i.e. each $v \in \mathit{Locals}(\pi')$, if $v \in \Delta(\pi')$ then $M'(v) = \text{zero}$.
3. For each formal parameter i.e. each p_i in $\mathit{Formals}(\pi') = [p_1, \ldots, p_k]$, if the corresponding actual parameter a_i is in $\Delta(\pi)$ then $M'(p_i) = M(a_i)$.

This definition leaves M unconstrained at formal parameters whose actual counterpart is not in $\Delta(\pi)$, which effects the required branching. The treatment of procedure returns is simpler: the signs of variables in the caller are the same as those before the call, except for the variable into which the result was returned, whose sign is read from the constraint in the callee.

$$succ_r(M, M', \pi, \pi', x, r, [a_1, \ldots, a_k]) \quad := \quad \oplus(M, x, M'(r))$$

There are many possible variations on the above sign analysis scheme, which we will not explore; we set out to provide a fully specified example of how to construct an analysis module, and we have done so.

3.4.3 Soundness conditions for analysis modules

In Subsection 3.4.1 we defined syntactically the interface for analysis modules and described informally the role of the interface's various components. But we still need to be assured that our analysis modules' functions return correct results, i.e. carry out a sound analysis. We now state conditions which ensure this.

In the background development of Chapter 2, soundness came from part 8 of Definition 2.3.4 (abstraction domain): we simply insisted that "if $s \in \gamma(a)$ then $f(i)(s) \subseteq \gamma(f(i)^{\#}(a))$". Now, however, there are three complicating issues:

1. In addition to intraprocedural statements, we must now deal with calls and returns.

2. We need to ensure that sharing is sound too, i.e. that information shared by the module really does describe the possible program states.

3. Rather than dealing with a tidy notional "transfer function" $f(i)$, we now have a list of 15 concrete rules for the execution of the various intraprocedural statement forms.

As a result, we end up with a list of 17 requirements for soundness; we have relegated the full list to Appendix A.3.

Definition 3.4.3. We say that an analysis module (as in Definition 3.4.2) is *sound* if it satisfies the conditions listed in Appendix A.3. □

A brief outline of the soundness conditions follows. For each intraprocedural statement form there are two conditions, the first saying that sharing is sound, and the second saying that successor computation is sound. Then there are three conditions concerning respectively initialisation, procedure call and procedure return.

As an example, consider the intraprocedural statement form `VarCopy`. The condition for sound sharing is:

sound-share-varcopy If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$

4. $s' = (e', h, A)$
5. $e' = e \oplus \{u \mapsto e(v)\}$

then $(s_0, s, s') \in [\![share(a, \texttt{VarCopy}(u, v), \pi)]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

Informally the rule can be read as follows. (We will disregard the state at procedure entry, s_0, for the sake of an easier description.) Premise 1. says that execution can reach location l in procedure π with state s, and then take one further intraprocedural step to location l' with state s'. Premise 2. says that the earlier state s is represented by the abstract value a. Premises 3. - 5. say that the states s and s' are related exactly as in the rule for execution of `Varcopy` statements. Finally the conclusion is that the formula produced by *share* from the abstract value a really does correctly describe the concrete states s and s' (and their relationship to each other).

The rule for sound successor computation is:

sound-succ-varcopy If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A)$
5. $e' = e \oplus \{u \mapsto e(v)\}$
6. $(s_0, s, s') \in [\![\Phi]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

then there exists $a' \in succ(a, \texttt{VarCopy}(u, v), \pi, \Phi)$ such that $(s_0, s') \in \gamma(a')$.

Again disregarding s_0, we give an informal explanation of the rule. Premises 1. - 5. are as before: execution reaches location l in π with state s and then, by mirroring

the execution of a `VarCopy` statement, proceeds to location l' with state s'. The earlier state s is represented by abstract value a.

The new premise 6. says that the incoming formula Φ (which in practise will consist of information shared by the other analysis modules) is a correct description of s, s' and their relationship to each other. This allows the *succ* function to *rely on* the information it receives. Finally the conclusion says that the concrete successor state s' is "covered" by at least one of the abstract values produced by *succ*.

We end this section with two brief remarks. Firstly, we originally wanted to put together our formalisation in a way that would deal with procedure calls and returns, and sharing of formulae, yet still allow the details of particular statements to be hidden behind the idea of a transfer function. Our attempts to construct such a formalisation were unsuccessful, however.

Secondly, there is a kind of "non-locality" in the soundness conditions, in the sense that the premise $\pi, s_0 \xrightarrow{l,s} l', s'$ demands that the location l in π actually be reachable with state s *in the whole program*. This is deliberate, because it allows an analysis module to perform some analysis of the whole program right at the beginning, and then refer to the results later when being asked for successors or for shared information.

For example, we later use this in Section 4.6, where we write a module which implements a simple type system. That module performs type inference for the whole program right at the beginning, and then shares information about the possible values of variables as the cooperative analysis proceeds.

3.5 Our module-based algorithm EXTRACT-MODEL

We are now in a position to present our algorithm EXTRACT-MODEL for automatically extracting sound models from programs; it is given in Algorithm Fragments 3.1

to 3.5. After some informal discussion of our algorithm, we will prove that it terminates and, provided the analysis module used is sound, produces sound models.

In the background development of Chapter 2, the forward propagation algorithm in Figure 2.5 was used to automatically extract models. The algorithm we give here is similar in that it uses forward propagation with a worklist, but with two main differences. Firstly we must now account for (recursive) procedures, which we do using summarisation. In this sense our algorithm is based on the summarisation algorithm from [RHS95]. The idea of summarisation is to "tabulate" the effect of each procedure, as a transformer from (abstract) entry states to return states; this information can then be reused across all call sites. With summarisation, it is not necessary to provide pre- and post-conditions for procedures. Secondly, because we are no longer hiding the details of intraprocedural statements behind transfer functions, the algorithm includes a case split, treating the various intraprocedural statements slightly differently. Our algorithm is generic in that it works with any analysis module provided.

Note that EXTRACT-MODEL is written to use a single analysis module; it does not include a mechanism for making multiple modules cooperate. This is because module combination and cooperation is done at the module level: we later provide a combinator $\diamond$ for *combining two modules into one*, which makes them cooperate by exchanging information. By iterated application of $\diamond$ we will be able to cooperatively combine as many modules as we like.

Our algorithm consists of three parts:

The worklist-step procedure (Algorithm Fragments 3.2 to 3.4) removes an abstract node from the worklist and calculates its abstract successors, adding them to the model. Successors that were not already present in the model are added to the worklist.

Conditional statements `If` Φ are dealt with (line 78) by reusing the mechanism

for sharing: for the 'if' branch we make a call to *succ* using `Skip` as the statement, and sending in the guard (adjusted for time) $\Phi[\mathfrak{C}\backslash\mathfrak{I}]$ as an extra piece of shared information. Similarly for the 'else' branch we send in the negated guard $\neg\Phi[\mathfrak{C}\backslash\mathfrak{I}]$.

Potentially dangerous statements (memory access or allocations) are also dealt with in this way. For example, for a `FieldRead` at variable v, the algorithm splits into two branches, one branch where the statement succeeds, and another branch where it fails. In this second branch we send in the formula $\neg \mathit{allocd}_{\mathfrak{I}}(v_{\mathfrak{I}})$.

The return-step procedure (Algorithm Fragment 3.5) processes procedure returns. Returns cannot be processed using the worklist because the same abstract state at a return point may need to be matched with several different abstract calling states, some of which may not yet have been generated.

To avoid repeated work, the variable *TriedInter* is used to track the pairs of abstract call and return nodes that have been dealt with already. Procedure summaries are implicit in the set N which is built to over-approximate the reachable (concrete) states.

The main procedure (Algorithm Fragment 3.1) does some initialisation and then performs worklist steps as long as they are possible, only trying return steps when the worklist is empty.

Algorithm Fragment 3.1 main procedure of EXTRACT-MODEL

```
    type  AbsNode = ProcNames × Locs × T

 5:                  ▷ (Here the type "X set" is inhabited by sets of values of type X.)

    vars  N, Worklist : AbsNode set
    vars  Edges, CallEdges, RtnEdges, TriedInter : (AbsNode × AbsNode) set
    vars  progress : boolean
10:
    procedure MAIN

        N, Worklist := {π1} × {start} × init(·)
        TriedInter := ∅
15:     Edges, CallEdges, RtnEdges := ∅
        progress := true

        while progress do
            progress := WORKLIST-STEP
20:         if progress = false then
                progress := RETURN-STEP
            end if
        end while

25: end procedure
```

Algorithm Fragment 3.2 worklist-step procedure of EXTRACT-MODEL (1 of 3)

```
    procedure WORKLIST-STEP
30:
        if there exists w, (π, l, a) such that
            w ∈ Worklist and (π, l, a) = w
        then
            Worklist := Worklist − {w}
35:         case Edges(Graph(π))(l) of

                ▷ First we handle the "ordinary" statements (those that can't go wrong)

            Skip : l' ⇒
                let
40:                 Succs := {π} × {l'} × succ(a, Skip, π, True)
                in
                    Edges := Edges ∪ ({(π, l, a)} × Succs)
                    Worklist := Worklist ∪ (Succs − N)
45:                 N := N ∪ Succs
                end let

            VarCopy(u, v) : l' ⇒
                let
50:                 Succs := {π} × {l'} × succ(a, VarCopy(u, v), π, True)
                in
                    Edges := Edges ∪ ({(π, l, a)} × Succs)
                    Worklist := Worklist ∪ (Succs − N)
                    N := N ∪ Succs
55:             end let

            AssignConst(u, k) : l' ⇒
                let
                    Succs := {π} × {l'} × succ(a, AssignConst(u, k), π, True)
60:             in
                    Edges := Edges ∪ ({(π, l, a)} × Succs)
                    Worklist := Worklist ∪ (Succs − N)
                    N := N ∪ Succs
                end let
65:

            Arith(u, v1, ⊗, v2) : l' ⇒
                let
                    Succs := {π} × {l'} × succ(a, Arith(u, v1, ⊗, v2), π, True)
                in
70:                 Edges := Edges ∪ ({(π, l, a)} × Succs)
                    Worklist := Worklist ∪ (Succs − N)
                    N := N ∪ Succs
                end let
```

Algorithm Fragment 3.3 worklist-step procedure of EXTRACT-MODEL (2 of 3)

```
75:
            If(Φ) : l'_1 : l'_2 ⇒
                let
                    TrueSuccs := {π} × {l'_1} × succ(a, Skip, π, Φ[𝔠\𝔍])
80:                 FalseSuccs := {π} × {l'_2} × succ(a, Skip, π, ¬Φ[𝔠\𝔍])
                    Succs := TrueSuccs ∪ FalseSuccs
                in
                    Edges := Edges ∪ ({(π, l, a)} × Succs)
                    Worklist := Worklist ∪ (Succs − N)
85:                 N := N ∪ Succs
                end let

            Choice : l'_1 : l'_2 ⇒
                let
90:                 Succs := {π} × {l'_1, l'_2} × succ(a, Skip, π, True)
                in
                    Edges := Edges ∪ ({(π, l, a)} × Succs)
                    Worklist := Worklist ∪ (Succs − N)
                    N := N ∪ Succs
95:             end let

        ▷ Next handle the "dangerous" statements (which access or allocate memory)

            FieldRead(u, v, f) : l' ⇒
100:            let
                    SuccessSuccs := {π} × {l'} × succ(a, FieldRead(u, v, f), π, true)
                    FailureSuccs := {π} × {memerror} × succ(a, Skip, π, ¬allocd_𝔍(v_𝔍))
                    Succs := SuccessSuccs ∪ FailureSuccs
                in
105:                Edges := Edges ∪ ({(π, l, a)} × Succs)
                    Worklist := Worklist ∪ (Succs − N)
                    N := N ∪ Succs
                end let

110:        FieldWrite(v, f, u) : l' ⇒
                let
                    SuccessSuccs := {π} × {l'} × succ(a, FieldWrite(v, f, u), π, true)
                    FailureSuccs := {π} × {memerror} × succ(a, Skip, π, ¬allocd_𝔍(v_𝔍))
                    Succs := SuccessSuccs ∪ FailureSuccs
115:            in
                    Edges := Edges ∪ ({(π, l, a)} × Succs)
                    Worklist := Worklist ∪ (Succs − N)
                    N := N ∪ Succs
                end let
120:
```

Algorithm Fragment 3.4 worklist-step procedure of EXTRACT-MODEL (3 of 3)

```
            New(u, v) : l' ⇒
                let
125:                SuccessSuccs := {π} × {l'} × succ(a, New(u, v), π, true)
                    FailureSuccs := {π} × {memerror} × succ(a, Skip, π, v_I ≤ 0)
                    Succs := SuccessSuccs ∪ FailureSuccs
                in
                    Edges := Edges ∪ ({(π, l, a)} × Succs)
130:                Worklist := Worklist ∪ (Succs − N)
                    N := N ∪ Succs
                end let

                                          ▷ Finally we do the interprocedural statements
135:
            Call(u, π', [p_1, ..., p_k]) : l' ⇒
                let
                    Succs := {π'} × {start} × succ_C(a, π, π', [p_1, ..., p_k])
                in
140:                CallEdges := CallEdges ∪ ({(π, l, a)} × Succs)
                    Worklist := Worklist ∪ (Succs − N)
                    N := N ∪ Succs
                end let

145:        Return(v) ⇒
                do nothing

        end case
        return true
150:  else
        return false
      end if
    end procedure

155:
```

Algorithm Fragment 3.5 return-step procedure of EXTRACT-MODEL

procedure RETURN-STEP

if there exists $(\pi, l, a), (\pi', l', a'), \hat{a}$ **such that**

160: $(\pi, l, a) \in N$ and $(\pi', l', a') \in N$

$Edges(Graph(\pi))(l) = \texttt{Call}(v, \pi', [p_1, \ldots, p_k]) : \hat{l}$

$Edges(Graph(\pi'))(l') = \texttt{Return}(v')$

$((\pi, l, a), (\pi', l', a')) \notin TriedInter$

$((\pi, l, a), (\pi', start, \hat{a})) \in CallEdge$

165: $((\pi', start, \hat{a}), (\pi', l', a')) \in (Edge)^*$

then

$TriedInter := TriedInter \cup \{((\pi, l, a), (\pi', l', a'))\}$

$Succs := \{\pi\} \times \{\hat{l}\} \times succ_R(a, a', \pi, \pi', v, v', [p_1, \ldots, p_k])$

$RtnEdges := RtnEdges \cup (\{(\pi', l', a')\} \times Succs)$

170: $Edges := Edges \cup (\{(\pi, l, a)\} \times Succs)$

$Worklist := Succs - N$

$N := N \cup Succs$

return *true*

else

175: **return** *false*

end if

end procedure

We now prove termination and soundness of our model-extraction algorithm.

Theorem 3.5.1. The EXTRACT-MODEL algorithm terminates.

Proof: For each procedure π, the set of control locations $Nodes(Graph(\pi))$ is finite, and thus

$$UsedLocs \quad := \quad \bigcup_{\pi \in Procs(P)} Nodes(Graph(\pi))$$

is also finite. Letting

$$M \quad := \quad \{\pi_1, \ldots, \pi_n\} \times UsedLocs \times T$$

we see that M is also finite. By inspection of the program, we see that it is an invariant of the main loop that $N \subseteq M$ and $TriedInter \subseteq M \times M$. Thus we use the following as a ranking function:

$$(\quad (|M| - |N|) + (|M \times M| - |TriedInter|), \quad |W| \quad)$$

using the usual lexicographic order on $\mathbb{N} \times \mathbb{N}$. We will show that this function strictly decreases in each iteration of the main loop except the last.

First consider what happens when a worklist step is performed. When successors are generated (e.g. *Succs* on line 41) consider the subset of these that are new ($Succs - N$). If this is empty, then the step reduces the size $|W|$ of the worklist (line 34) while leaving N and *TriedInter* unchanged. On the other hand if $Succs - N$ is nonempty, then the size of N increases with *TriedInter* again unchanged.

Secondly consider what happens when a return step is performed: *TriedInter* gains one new element, and the size of N stays the same or increases. □

Theorem 3.5.2. The EXTRACT-MODEL algorithm produces sound models, provided the analysis module used is sound.

Proof: We check the conditions sound-init, sound-intra, sound-call and sound-

return in that order. Here when we write N we mean the contents of N upon termination of the algorithm, unless otherwise specified; similarly for the edge relations. Because there are many cases to check for intraprocedural statements, and because the argument is very similar for each case, we just check four here.

sound-init: Using Definition 3.4.3 (sound-m-init part), there exists $a \in \mathit{init}(\cdot)$ such that $(s^{\mathit{start}}, s^{\mathit{start}}) \in \gamma(a)$. On line 13 of the algorithm $(\pi_1, \mathit{start}, a')$ is added to N.

sound-intra: Let $\pi, s_0 \xrightarrow{l,s} l', s'$, and let N contain an element $w = (\pi, l, a)$ with $(s_0, s) \in \gamma(a)$. Inspection of the algorithm shows that whenever a new element is added to N, it is also added to W, and since $W = \emptyset$ when the algorithm terminates, w must at a later stage be removed from W (on line 34) and processed in a worklist step. It is also easy to see that during the execution of the algorithm the set N and the edge relations only increase, so it will suffice to show that an appropriate node (π, l', a') and an appropriated edge were added at some stage. We split our analysis into cases depending on which of the intraprocedural rules was used to complete the derivation:

skip: From the premises of the skip rule we have $s = s'$ and $\pi, s_0 \xrightarrow{p} l, s'$ and $\mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Skip} : l'$.

When (π, l, a) is removed from W the case on line 41 is invoked. By Definition 3.4.3 (sound-succ-skip part), there exists $a' \in \mathit{succ}(a, \texttt{Skip}, \pi, \mathit{true})$ such that $(s_0, s') \in \gamma(a)$. To finish we note that (π, l', a') will be added to N on line 45 and that an edge $((\pi, l, a), (\pi, l', a'))$ will be added to *Edges* on line 43.

varcopy: From the premises of the varcopy rule we have $\pi, s_0 \xrightarrow{p} l, s$ and $\mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{VarCopy}(u, v) : l'$ where: $s = (e, h, A)$, $s' = (e', h, A)$ and $e' = e \oplus \{u \mapsto e(v)\}$.

When (π, l, a) is removed from W the case on line 50 is invoked. By

Definition 3.4.3 (sound-succ-varcopy part), there exists $a' \in succ(a, \texttt{VarCopy}(u, v), \pi, true)$ such that $(s_0, s') \in \gamma(a)$. To finish we note that (π, l', a') will be added to N on line 54 and that an edge $((\pi, l, a), (\pi, l', a'))$ will be added to *Edges* on line 52.

fieldwrite-success: From the premises of the fieldwrite-success rule we have $\pi, s_0 \xrightarrow{p} l, s$ and $Edges(Graph(\pi))(l) = \texttt{FieldWrite}(v, f, u) : l'$ where: $s = (e, h, A)$, $s' = (e, h', A)$, $e(v) \in A$ and $h' = h \oplus \{(f, e(v)) \mapsto e(u)$.

When (π, l, a) is removed from W the case on line 112 is invoked. By Definition 3.4.3 (sound-succ-fieldwrite part), there exists $a' \in succ(a, \texttt{FieldWrite}(v, f, u), \pi, true)$ such that $(s_0, s') \in \gamma(a')$. To finish we note that (π, l', a') will be added to N on line 118 and that an edge $((\pi, l, a), (\pi, l', a'))$ will be added to *Edges* on line 116.

fieldwrite-failure: From the premises of the fieldwrite-failure rule we have $\pi, s_0 \xrightarrow{p} l, s$ and $Edges(Graph(\pi))(l) = \texttt{FieldWrite}(v, f, u) : l'$ where: $s = (e, h, A)$ and $e(v) \notin A$. From the semantics of $\mathscr{L}$ and the fact that $e(v) \notin A$, we see that $(s_0, s, s') \in [\![\neg allocd_{\mathbf{I}}(v_{\mathbf{I}})]\!]$.

When (π, l, a) is removed from W the case on line 112 is invoked. By Definition 3.4.3 (sound-succ-skip part), there exists $a' \in succ(a, \texttt{Skip}, \pi, \neg allocd_{\mathbf{I}}(v_{\mathbf{I}}))$ such that $(s_0, s') \in \gamma(a')$. To finish we note that (π, l', a') will be added to N on line 118 and that an edge $((\pi, l, a), (\pi, l', a'))$ will be added to *Edges* on line 116.

other intraprocedural rules: The cases for the remaining intraprocedural rules are similar to the preceding four cases; we omit these "variations on a theme".

sound-call: Let $\pi, s_0 \rightarrow l, s : \pi', s'$ and let N contain an element (π, l, a) with $(s_0, s) \in \gamma(a)$. The conclusion $\pi, s_0 \rightarrow l, s : \pi', s'$ can only have been derived using the call-1 rule, and from its premises we have $\pi, s_0 \xrightarrow{p} l, s$ and

$Edges(Graph(\pi))(l) = \texttt{Call}(u, \pi', [p_1, \ldots, p_k]) : l'$, where: $s = (e, h, A)$, $s' = (e', h, A)$ and

$$e'(x) = \begin{cases} e(p_i) & \text{if } x \text{ is } f_i \\ 0 & \text{otherwise} \end{cases}$$

where $[f_1, \ldots, f_j] = Formals(\pi')$. When (π, l, a) is removed from W the case on line 138 is invoked. By Definition 3.4.3 (succ-call part), there exists $a' \in succ_C(a, \pi, \pi', [p_1, \ldots, p_k])$ such that $(s', s') \in \gamma(a')$. To finish we note that (π', l', a') will be added to N on line 142 and that an edge $((\pi, l, a), (\pi', l', a'))$ will be added to *CallEdges* on line 140.

sound-return: Lastly we prove the sound-return part. So let $\pi, s_0 \xrightarrow{l_1, s_1, \bar{\pi}, s_2, l_3, s_3} l', s'$ and let N contain elements (π, l_1, a_1), $(\bar{\pi}, start, a_2)$, $(\bar{\pi}, l_3, a_3)$ such that:

$(s_0, s_1) \in \gamma(a_1)$

$(s_2, s_3) \in \gamma(a_3)$

$((\pi, l_1, a_1), (\bar{\pi}, start, a_2)) \in CallEdges$

$((\bar{\pi}, start, a_2), (\bar{\pi}, l_3, a_3)) \in Edges^*$

Now $\pi, s_0 \xrightarrow{l_1, s_1, \bar{\pi}, s_2, l_3, s_3} l', s'$ can only be obtained by the return rule, so from its premises we see that: $Edges(Graph(\pi))(l_1)$ and $Edges(Graph(\bar{\pi}))(l_3)$ have respectively the forms $\texttt{Call}(v, \bar{\pi}, [p_1, \ldots, p_k]) : l'$ and $\texttt{Return}(v')$.

Hence we see that on line 159 the selection $(\pi, l_1, a_1), (\bar{\pi}, l_3, a_3), a_2$ must eventually be made: the selection must eventually become "eligible", after which it cannot be put off forever because when the algorithm terminates there are no more possible selections.

Also from the premises of the return rule we have

$\pi, s_0 \rightarrow l_1, s_1 : \bar{\pi}, s_2$

$\bar{\pi}, s_2 \xrightarrow{\bar{p}} l_3, s_3.$

$s_1 = (e_1, h_1, A_1)$

$$s_3 = (e_3, h_3, A_3)$$

$$s' = (e', h_3, A_3)$$

$$e' = e_1 \oplus \{u \mapsto e_3\,(v)\}$$

Only the call-1 rule can produce $\pi, s_0 \rightarrow l_1, s_1 : \bar{\pi}, s_2$, and from its premises we have: $s_2 = (e_2, h_1, A_1)$ where

$$e_2\,(x) = \begin{cases} e_1\,(p_i) & \text{if } x \text{ is } f_i \\ 0 & \text{otherwise} \end{cases}$$

with $[f_1, \ldots, f_j] = \mathit{Formals}\,(\pi')$.

Finally, then, we have collected all the conditions needed to invoke Definition 3.4.3 (succ-return part), which shows that there exists $a' \in \mathit{succ}_R(a_1, a_3, \pi, \bar{\pi}, u, v, [p_1, \ldots, p_k])$ such that $(s_0, s') \in \gamma(a')$. On line 172 (π, l', a') will be added to N, and on lines 169 and 170 the required edges $((\bar{\pi}, l_3, a), (\pi, l', a')) \in \mathit{ReturnEdges}$ and $((\pi, l_1, a_1), (\pi, l', a')) \in \mathit{Edges}$ will be added. □

Running example:

Running the above algorithm EXTRACT-MODEL with **compsigns** $\langle\Delta\rangle$, the sign analysis module from Subsection 3.4.2, with configuration

$$\Delta \quad := \quad \{\mathit{main} \mapsto \{x, y\}, \quad \mathit{chooseNat} \mapsto \{n, \mathit{one}\}, \quad \mathit{intSqrt} \mapsto \{n, \mathit{one}, x\}\}$$

produces exactly the model depicted in Figure 3.10 (page 104).

3.6 Combining analysis modules

3.6.1 The module combinator $\diamond$

When introducing our model extraction algorithm in the previous section, we promised to provide a combinator $\diamond$ that combines two analysis modules and makes them cooperate by exchanging information, using $\mathscr{L}$ as a common language. The result of applying $\diamond$ to two modules is another module, which can be combined again, or used in the analysis algorithm. We now deliver this $\diamond$.

Definition 3.6.1. Given modules $\mathbf{D}$ and $\mathbf{E}$, we construct their *combination* $\mathbf{D}\diamond\mathbf{E}$ as:

$$(\mathbf{D}\diamond\mathbf{E}).T := \begin{array}{ll} & \mathbf{D}.T \\ \times & \mathbf{E}.T \end{array}$$

$$(\mathbf{D}\diamond\mathbf{E}).\gamma\,((d,e)) := \begin{array}{ll} & \mathbf{D}.\gamma\,(d) \\ \cap & \mathbf{E}.\gamma\,(e) \end{array}$$

$$(\mathbf{D}\diamond\mathbf{E}).\mathit{share}((d,e),s,\pi) := \begin{array}{ll} & \mathbf{D}.\mathit{share}(d,s,\pi) \\ \wedge & \mathbf{E}.\mathit{share}(e,s,\pi) \end{array}$$

$$(\mathbf{D}\diamond\mathbf{E}).\mathit{succ}((d,e),s,\pi,\Phi) := \begin{array}{ll} & \mathbf{D}.\mathit{succ}(d,s,\Theta) \\ \times & \mathbf{E}.\mathit{succ}(e,s,\Theta) \end{array}$$

$$\text{where } \Theta \triangleq \Phi \wedge (\mathbf{D}\diamond\mathbf{E}).\mathit{share}((d,e),s,\pi)$$

$$(\mathbf{D}\diamond\mathbf{E}).\mathit{succ}_C\left(\begin{array}{l}(d,e),\pi,\pi',\\ [a_1,\ldots,a_k]\end{array}\right) := \begin{array}{ll} & \mathbf{D}.\mathit{succ}_C(d,\pi,\pi',[a_1,\ldots,a_k]) \\ \times & \mathbf{E}.\mathit{succ}_C(e,\pi,\pi',[a_1,\ldots,a_k]) \end{array}$$

$$(\mathbf{D}\diamond\mathbf{E}).\mathit{succ}_R\left(\begin{array}{l}(d,e),(d',e'),\\ \pi,\pi',x,r,\\ [a_1,\ldots,a_k]\end{array}\right) := \begin{array}{ll} & \mathbf{D}.\mathit{succ}_R(d,d',\pi,\pi',x,r,[a_1,\ldots,a_k]) \\ \times & \mathbf{E}.\mathit{succ}_R(e,e',\pi,\pi',x,r,[a_1,\ldots,a_k]) \end{array}$$

$$(\mathbf{D}\diamond\mathbf{E}).\mathit{init}(\cdot) := \begin{array}{ll} & \mathbf{D}.\mathit{init}(\cdot) \\ \times & \mathbf{E}.\mathit{init}(\cdot) \end{array}$$

□

When asked to share a formula, $\mathbf{D} \diamond \mathbf{E}$ asks each of $\mathbf{D}$ and $\mathbf{E}$ for a formula, and conjoins the results. When generating successors, $\mathbf{D}$ and $\mathbf{E}$ are each given the other's shared formulae, as well as the incoming formula Φ, and asked to generate their own sets of successors. All pairs of these are then returned. We do not try to eliminate inconsistent pairs here, because this happens in a later iteration of the analysis anyway, when those pairs have their own successors generated.

Remark 3.6.2. If $\mathbf{D}$ and $\mathbf{E}$ are sound modules (as per Definition 3.4.3) then so is their combination $\mathbf{D} \diamond \mathbf{E}$. □

Intuitively, it seems from Definition 3.6.1 that, when combining $n > 1$ modules with $\diamond$, it doesn't matter in which order or which way round we combine them. We believe that it is possible to give an appropriate notion of equivalence between modules, whence our combination operator $\diamond$ can be shown to be commutative, associative and idempotent, thereby making the space of modules into a semilattice. Further, we suspect that the ordering in this semilattice coincides with a natural notion of refinement, where modules which are higher in the refinement order produce tighter analysis results.

3.6.2 Example of combination: obtaining a "thorough" sign analysis

To illuminate the definition of $\diamond$, we will introduce another module, called the consistency checking module, and explore how it combines beneficially with our sign analysis module. In fact, we will see that by combining in this way, we automatically upgrade our sign analysis (which is "compositional") to a more precise kind (called "thorough").

The idea of the consistency checking module is that it doesn't keep any constraints of its own concerning the program state — it simply checks for consistency the

information it receives via sharing of formulae. This checking is done using a theorem prover ρ; hence strictly we construct an indexed family $\mathbf{cons}\,\langle\rho\rangle$ of modules.

Because the module maintains no constraints of its own, the set T of abstract values consists of a single element Um. We call this element Um because, like the spoken utterance "um", it conveys no information:

$$\begin{aligned} T &:= \{\text{Um}\} \\ \gamma(\text{Um}) &:= \mathit{State} \times \mathit{State} \end{aligned}$$

The important component for this module, which performs the actual consistency checking, is the function *succ*:

$$\mathit{succ}(a, s, \pi, \Phi) := \text{if } \rho \text{ proves } \neg\Phi \text{ then } \emptyset \text{ else } \{\text{Um}\}$$

The other components effectively do nothing:

$$\begin{aligned} \mathit{share}(a, s, \pi) &:= \mathit{true} \\ \mathit{succ}_C(a, \pi, \pi', [a_1, \ldots, a_k]) &:= \{\text{Um}\} \\ \mathit{succ}_R(a, a', \pi, \pi', x, r, [a_1, \ldots, a_k]) &:= \{\text{Um}\} \\ \mathit{init}(\cdot) &:= \{\text{Um}\} \end{aligned}$$

Running example:

Earlier we applied our model extraction algorithm to our example program from Figure 3.1, using the sign analysis module $\mathbf{compsigns}\,\langle\Delta\rangle$ from Subsection 3.4.2, with the particular configuration Δ given on page 134. This produced the model in Figure 3.10 (page 104).

Now we apply the algorithm to the same program, but this time using the combined module $\mathbf{compsigns}\,\langle\Delta\rangle \diamond \mathbf{cons}\,\langle\rho\rangle$. This produces a different, more precise model,

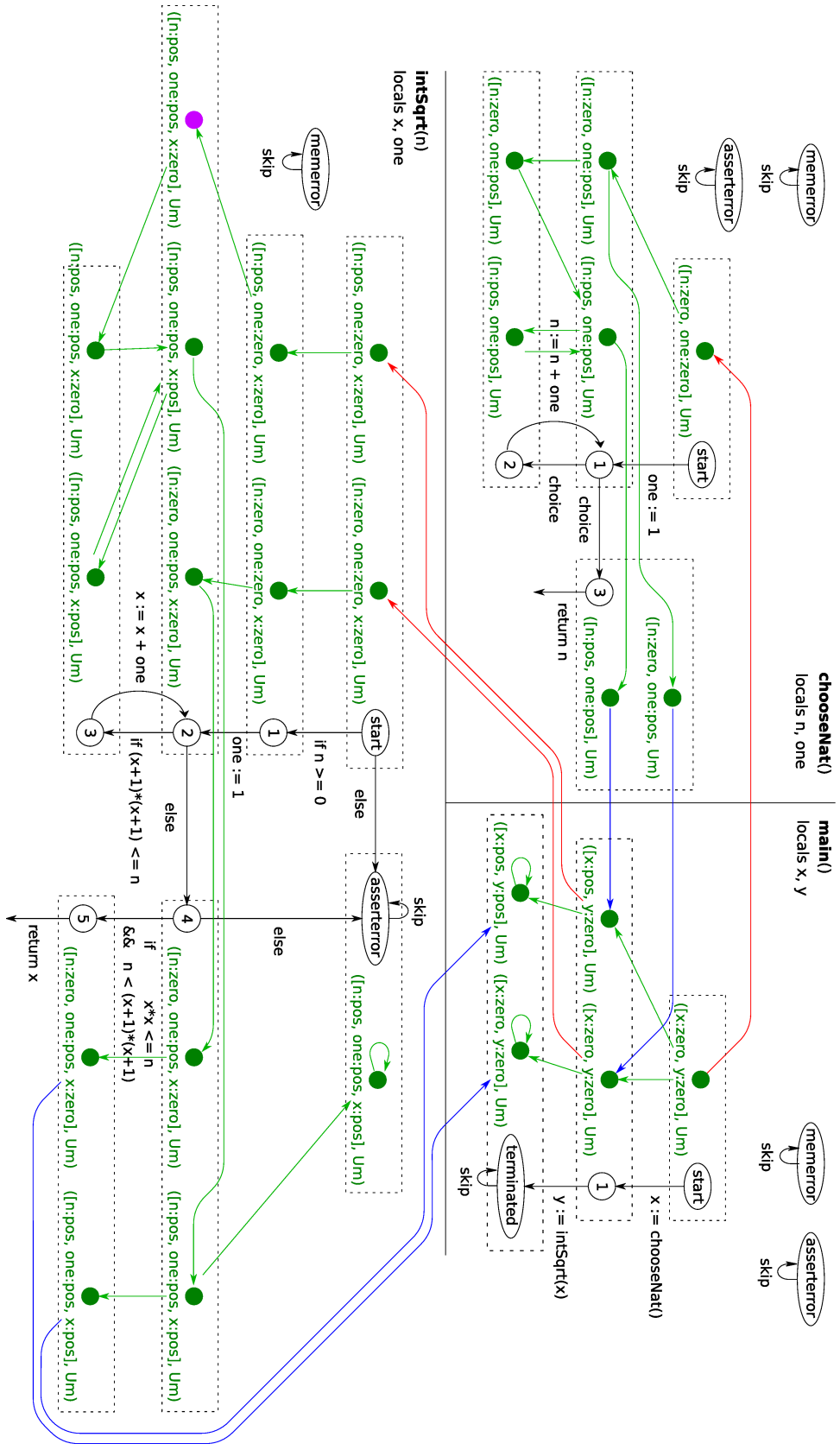

Figure 3.11: Another abstract model, more precise than the one in Figure 3.10, built using the combined domain **compsigns** $\langle\Delta\rangle \diamond$ **cons** $\langle\rho\rangle$ which effects a thorough sign analysis. (The relations *Edges*, *CallEdges* and *ReturnEdges* are shown in green, red and blue respectively. This graph is hand-edited for better layout, though our implementation can generate similar graphs; see Chapter 4.)

shown in Figure 3.11. Each abstract node is now labelled with a *pair* of values, one from the sign analysis module and one form the consistency checking module. (Of course, the component from the consistency checking module is always Um.) Let us see how sharing between the modules leads to a more precise model.

Consider the abstract node labelled with

$$([n : \text{pos}, \mathit{one} : \text{pos}, x : \text{zero}], \text{Um})$$

at location 2 of procedure intSqrt (i.e. the node shown in purple in Figure 3.11). For ease of layout we will denote by M the sign analysis component at this abstract node. Let us trace the generation of this node's successors for the 'else' branch.

Example 3.6.3. The relevant invocation of $(\mathbf{compsigns}\,\langle\Delta\rangle \diamond \mathbf{cons}\,\langle\rho\rangle).\mathit{succ}$ is:

$$(\mathbf{compsigns}\,\langle\Delta\rangle \diamond \mathbf{cons}\,\langle\rho\rangle).\mathit{succ}\Big((M, \text{Um}),\ \texttt{Skip},\ \text{intSqrt},\ \neg((x_{\mathbf{I}}{+}1){\times}(x_{\mathbf{I}}{+}1) \leq n_{\mathbf{I}})\Big)$$

Definition 3.6.1 shows how to compute the above. We break the computation down into four stages.

1. **Collect shared information** First we invoke $(\mathbf{compsigns}\,\langle\Delta\rangle{\diamond}\mathbf{cons}\,\langle\rho\rangle).\mathit{share}$ to gather up the shared information provided by the two modules:

$$\begin{array}{lll} & (\mathbf{compsigns}\,\langle\Delta\rangle \diamond \mathbf{cons}\,\langle\rho\rangle).\mathit{share}((M, \text{Um}), \texttt{Skip}, \text{intSqrt}) \\ \rightsquigarrow & \mathbf{compsigns}\,\langle\Delta\rangle\,.\mathit{share}(M, \texttt{Skip}, \text{intSqrt}) & \\ & \quad \wedge \quad \mathbf{cons}\,\langle\rho\rangle\,.\mathit{share}(\text{Um}, \texttt{Skip}, \text{intSqrt}) & \\ \rightsquigarrow & \mathbf{compsigns}\,\langle\Delta\rangle\,.\mathit{share}(M, \texttt{Skip}, \text{intSqrt}) \quad \wedge \quad \mathit{True} & \\ \rightsquigarrow & \mathbf{compsigns}\,\langle\Delta\rangle\,.\mathit{share}(M, \texttt{Skip}, \text{intSqrt}) & \\ \rightsquigarrow & n_{\mathbf{I}} > 0 \quad \wedge \quad \mathit{one}_{\mathbf{I}} > 0 \quad \wedge \quad x_{\mathbf{I}} = 0 & \end{array}$$

and then conjoin this to the formula passed into $(\mathbf{compsigns}\,\langle\Delta\rangle{\diamond}\mathbf{cons}\,\langle\rho\rangle).\mathit{succ}$;

for ease of layout we call the result Θ.

$$\Theta \quad := \quad \neg((x_{\mathbf{I}}+1)\times(x_{\mathbf{I}}+1)\leq n_{\mathbf{I}}) \quad \wedge \quad n_{\mathbf{I}}>0 \quad \wedge \quad one_{\mathbf{I}}>0 \quad \wedge \quad x_{\mathbf{I}}=0$$

2. Compute successors for first module (sign analysis) Next we invoke the successor functions from each module, supplying Θ as the extra information parameter. For the sign analysis module, the *compositional*, three-valued evaluation of the formula is unable to determine the truth value of Θ with respect to M, i.e. $\mathit{check}(\Theta, M)$ returns 'unknown'. The important part of the evaluation of $\mathit{check}(\Theta, M)$ is:

$$\begin{aligned}
& M((x_{\mathbf{I}}+1)\times(x_{\mathbf{I}}+1)) \leq M(n_{\mathbf{I}}) \\
\rightsquigarrow\ & M(x_{\mathbf{I}}+1)\times M(x_{\mathbf{I}}+1)\leq \mathrm{pos} \\
\rightsquigarrow\ & (M(x_{\mathbf{I}})+M(1))\times(M(x_{\mathbf{I}})+M(1))\leq \mathrm{pos} \\
\rightsquigarrow\ & (M(x_{\mathbf{I}})+\mathrm{pos})\times(M(x_{\mathbf{I}})+\mathrm{pos})\leq \mathrm{pos} \\
\rightsquigarrow\ & (\mathrm{zero}+\mathrm{pos})\times(\mathrm{zero}+\mathrm{pos})\leq \mathrm{pos} \\
\rightsquigarrow\ & \mathrm{pos}\times\mathrm{pos}\leq \mathrm{pos} \\
\rightsquigarrow\ & \mathrm{pos}\leq \mathrm{pos} \\
\rightsquigarrow\ & \mathrm{unknown}
\end{aligned}$$

Being conservative, the sign analysis module treats unknown formulae as true, and returns the set of successors $\{M\}$:

$$\begin{aligned}
& \mathbf{compsigns}\,\langle\Delta\rangle\,.\mathit{succ}(M, \mathtt{Skip}, \mathrm{intSqrt}, \Theta) \\
\rightsquigarrow\ & \begin{cases} \emptyset & \text{if } \mathit{check}(\Phi, M) = \mathit{false} \\ \mathit{succ}'(M, \mathtt{Skip}, \mathrm{intSqrt}, \Theta & \text{otherwise} \end{cases} \\
\rightsquigarrow\ & \mathit{succ}'(M, \mathtt{Skip}, \mathrm{intSqrt}, \Theta) \\
\rightsquigarrow\ & \{M\}
\end{aligned}$$

3. Compute successors for second module (consistency checking) Next we invoke the successor function for the consistency checking module. This causes the formula $\neg\Theta$ to be sent to the theorem prover, and *it is here that the shared information provided by the sign analysis module is put to good use.* A reasonable theorem prover will prove $\neg\Theta$, perhaps working as follows (in each step rewriting to an equivalent or stronger formula):

$$
\begin{array}{lll}
 & \neg\Theta & \\
\rightsquigarrow & (x_{\mathfrak{I}}+1)\times(x_{\mathfrak{I}}+1)\leq n_{\mathfrak{I}} \quad \vee \quad n_{\mathfrak{I}}\leq 0 \quad \vee \quad \mathit{one}_{\mathfrak{I}}\leq 0 \quad \vee \quad \neg x_{\mathfrak{I}}=0 & \\
\rightsquigarrow & (x_{\mathfrak{I}}+1)\times(x_{\mathfrak{I}}+1)\leq n_{\mathfrak{I}} \quad \vee \quad n_{\mathfrak{I}}\leq 0 \quad \vee \quad \neg x_{\mathfrak{I}}=0 & \\
\rightsquigarrow & x_{\mathfrak{I}}=0 \quad \rightarrow \quad \Big((x_{\mathfrak{I}}+1)\times(x_{\mathfrak{I}}+1)\leq n_{\mathfrak{I}} \quad \vee \quad n_{\mathfrak{I}}\leq 0\Big) & \\
\rightsquigarrow & x_{\mathfrak{I}}=0 \quad \rightarrow \quad \Big(1\leq n_{\mathfrak{I}} \quad \vee \quad n_{\mathfrak{I}}\leq 0\Big) & \\
\rightsquigarrow & 1\leq n_{\mathfrak{I}} \quad \vee \quad n_{\mathfrak{I}}\leq 0 &
\end{array}
$$

So we obtain:

$$
\begin{array}{ll}
 & \mathbf{cons}\,\langle\rho\rangle\,.\mathit{succ}(\mathrm{Um},\mathtt{Skip},\mathrm{intSqrt},\Theta) \\
\rightsquigarrow & \text{if } \rho \text{ proves } \neg\Theta \text{ then } \emptyset \text{ else } \{\mathrm{Um}\} \\
\rightsquigarrow & \emptyset
\end{array}
$$

4. Form the Cartesian product of the results The final step is to form the Cartesian product of the two sets of successors.

$$(\mathbf{compsigns}\,\langle\Delta\rangle \diamond \mathbf{cons}\,\langle\rho\rangle).succ \left(\begin{array}{l} (M, \mathrm{Um}), \quad \texttt{Skip}, \quad \mathrm{intSqrt}, \\ \neg((x_{\mathbf{I}} + 1) \times (x_{\mathbf{I}} + 1) \leq n) \end{array} \right)$$

$$\rightsquigarrow \quad \mathbf{compsigns}\,\langle\Delta\rangle\,.succ(M, \texttt{Skip}, \mathrm{intSqrt}, \Theta)$$

$$\times\, \mathbf{cons}\,\langle\rho\rangle\,.succ(\mathrm{Um}, \texttt{Skip}, \mathrm{intSqrt}, \Theta)$$

$$\rightsquigarrow \quad \{M\} \times \emptyset$$

$$\rightsquigarrow \quad \emptyset$$

Informally, here we see that, using the shared information provided by the sign analysis module, the consistency checking module has found that the guard of the 'else' branch is not consistent with what is known about the program state, and thus returned an empty set of successors. When a number of modules have been combined, if *any one of them* detects an inconsistency, then there will be no successors in the combined module. □

So what happened in the preceding example? Why did the consistency checking module detect the inconsistency, but the sign analysis module did not? The answer is that the function *check*, which we call compositional because it works by recursion on the structure of formulae, loses precision. On the other hand, by using a theorem prover, the consistency checking module performs a "thorough" check of the formula which, while more expensive than the compositional check, is more precise.

We have not seen any existing research that distinguishes between a thorough and compositional sign analysis, but this distinction seems natural to us, because it has been previously observed in other situations where three-valued semantics are used, such as in [RLS02] with propositional logic, in [AH06] with CTL ∩ LTL and in [BG00, GH05] with other temporal logics. Appendix A.1.1 contains a few more details.

Comparing the resulting models Comparing the models in Figures 3.10 and 3.11 (pages 104 and 138 respectively) we can see that using the combined analysis, i.e. the thorough sign analysis, produces a model that is more precise (and thus better for verification), because one spurious branch of abstract nodes is eliminated (starting from the purple node).

In terms of computational cost, the cost *per successor computation* is much higher with the thorough sign analysis due to the extra invocation of the theorem prover; on the other hand, with the thorough analysis *fewer successor computations were needed* in building the model, because one spurious branch of abstract nodes was eliminated.

In any case we won't claim, and have no need to claim, that the thorough analysis is "better" than the compositional one or vice versa; the users of our verification system can make that decision for themselves. But we have provided a simple method for obtaining one analysis from the other via an *ad-hoc* combination; we believe that this supports the user's ability to experiment with different kinds of analysis. Our consistency checking module could be reused unmodified, for instance, to automatically upgrade other analyses, such as module based on the parity abstraction.

Further remarks We end this section with some further remarks concerning our preceding example of module combination and the benefits of information sharing.

Firstly, we should perhaps have reservations about our use of the term "thorough", because the logic $\mathscr{L}$ is undecidable and therefore we cannot obtain a check that is "as thorough as possible"; our thoroughness is relative to the completeness of the theorem prover ρ. However, if the user wants to try a different theorem prover ρ', or even combine two or three of them, this is easily possible: simply form e.g.

$$\mathbf{compsigns}\,\langle\Delta\rangle \;\diamond\; \mathbf{cons}\,\langle\rho\rangle \;\diamond\; \mathbf{cons}\,\langle\rho'\rangle \;\diamond\; \mathbf{cons}\,\langle\rho''\rangle$$

Secondly, in case the reader is wondering why we did not introduce the consistency checking domain in Chapter 2 (Background), the reason is that there is *no such domain* in the conventional settings. The module provides reasoning power but no real abstract state of its own, and this only makes sense in the context of formula sharing.

Thirdly, because the sign domain is "flat", the only improvement in precision that can be achieved by sharing formulae is to detect inconsistency. With non-flat domains, more subtle improvements can occur.

3.7 Discussion of our choice of common logic $\mathscr{L}$

We finish this chapter by discussing our decision to use a first order logic with transitive closure, or FO(TC), as our common logic $\mathscr{L}$. Essentially this decision is an educated guess, and we offer the following arguments in support of it:

1. **Some notion of reachability is clearly necessary**. It has been observed (e.g. [IRR+04, BCO04]) that the ability to express the reachability of data via particular variables and fields is essential for analysing and verifying programs using dynamic data structures[1]. FO(TC) includes a very general notion of reachability.

2. **Sound FO(TC) reasoning can be done with existing first order provers**. By encoding transitive closure subformulae using first order predicates (see page 63), one can perform sound (but necessarily incomplete) reasoning about FO(TC) formulae using the use of the wide variety of existing first order provers.

3. **FO(TC) can express some structural/spatial reasoning**. In [BCO04], decidable fragments of separation logic are given for reasoning about lists

[1] In this light, however, [MN05] is interesting because it shows what can still be said about data structures without using reachability.

and trees. Given the nature of separation logic one might expect these to be "intrinsically second order", but in fact the decidable fragments from [BCO04] (and more) can be translated into a decidable fragment of FO(TC) (this fact is briefly mentioned in [YRS+06]).

4. **Ownership appears related to transitive closure.** The ownership concept mentioned in Subsection 2.5.3, which has been proposed as a crucial concept in making the verification of object-oriented programs scalable, seems to be related to transitive closure because it talks about paths through the heap. Suppose O_1 and O_2 are objects at the top level of a program, and that O_2 owns an object P. Then the owner-as-dominator formulation of ownership says that P must be unreachable from O_1, except along paths which go via O_2. This means that O_1 can only access P by going through P's owner O_2.

In any case, our development depends only on very minimal properties[2] of $\mathscr{L}$, and thus the common language can be changed freely.

What other languages might we use? Second order logic, or even full higher order logic, spring to mind. These would certainly allow greater expressiveness, and would ease the problem of getting information from the modules into the intermediate language; however transfer in the other direction would become correspondingly more difficult. There appears to be little known about automated theorem proving in second and higher order logics. Technically, sound reasoning about higher order logics can be encoded in first order logic (see e.g. [NR03]) so an analogue of point 2. above applies, but the encodings appear too awkward to use, whereas the encodings of FO(TC) are quite clean.

[2]Specifically: $\mathscr{L}$ is required to contain *True* and the conjunction operator $\wedge$, and be able to express the facts that a program variable is nonnegative, and contains an allocated heap address.

3.8 Summary

In this chapter we presented our new modular verification framework, based on the ideas of the open product from Chapter 2. In our approach, a number of independently developed software modules called *analysis modules* are "plugged into" a generic verification algorithm, which invokes each analysis module as necessary to analyse the target program. Each analysis module implements a different kind of abstraction, and, as the analysis proceeds, the generic algorithm controls the propagation of information between them. We gave a detailed description of all aspects of our system: the kind of programs to be analysed, the kind of assertions to be checked, the interface which analysis modules implement, the common language with which they communicate, and the generic verification algorithm. We further gave a formal account of all these aspects of the framework, culminating in a proof that the verification results produced are sound.

The important parts of the development were as follows.

- The target programs, i.e. the programs analysed by the verification system, come from an idealised imperative heap-manipulating language with recursive procedures and non-determinism. Programs are represented in CFG form.
- Intuitively an analysis module is a program analysis/verification tool which has been appropriately wrapped for integration into our system. Analysis modules implement a common interface, which extends that of an abstraction domain, adding functions for the propagation of information between modules.
- For the common language with which the analysis modules communicate, we chose a first order logic with transitive closure, or FO(TC). FO(TC) is more expressive than first order logic (which cannot express important properties of data structures e.g. reachability), but avoids the complications of second order or higher order logic.

- The generic verification algorithm is worklist-based and uses procedure summarisation to handle recursive procedures, without requiring that they be annotated with pre- and post-conditions. The algorithm is initially presented in a form that uses a single analysis module, and is then extended to arbitrary numbers of analysis modules via the $\diamond$ operator, which, given two analysis modules, procedures their cooperating combination.

We showed how two particular analysis modules — for the compositional sign analysis and consistency checking techniques — can be built, and showed how, by exchanging formulae, they cooperate to produce better results.

Having set out our verification framework, in the next chapter we will consider the task of implementing the framework effectively.

Chapter 4

Implementation: the HECTOR system

In this chapter we describe our experimental tool HECTOR which implements the verification framework developed in Chapter 3, beginning with an overall summary of the implementation.

4.1 Overview of implementation

Our system HECTOR comprises:

- **Module-based verification framework:** The core of HECTOR is our module-based model extraction algorithm (the "broker", Section 3.5), implemented for any number of analysis modules combined cooperatively using the $\diamond$ operator (Subsection 3.6.1). We include code for identifying counterexample paths when verification fails.
- **Analysis modules:** Seven techniques (listed shortly) are implemented as analysis modules, with a mix of deep but expensive and cheap but shallow analyses.

- **Visualisation facilities (with web interface):** HECTOR can produce visual output showing: control flow graphs, abstract models of programs (or parts thereof), and counterexamples to verification. These outputs can be customised in various ways, such as by hiding the components from a particular analysis module, and can optionally include an indication of where sharing between modules proved to be useful. For ease of use, the visualisation facilities are accessed through a web interface, which can also be used to organise models into folders and annotate them with comments.

- **Model checking extensions:** After developing the basic HECTOR system we added three interesting extensions: model checking of a "two-level" version of safety LTL, falsification of (some) safety properties as well as their verification, and the post-pruning of models, which cheaply prunes away (some) infeasible states. The discussion of these extensions is deferred until Chapter 6.

The analysis modules currently implemented are:

tpa provides trivector predicate abstraction as in Subsection 2.5.1.

mpa provides monomial predicate abstraction also as in Subsection 2.5.1.

tvla provides three-valued shape analysis as on page 65 in Subsection 2.5.3.

types (Section 4.6) places a type system on top of our untyped language, performing type inference to discover variables that are non-negative, allocated, Boolean or null.

symbprop (Section 4.7) provides *symbolic constant propagation* [Min06, LF08].

compsigns (Section 4.3) provides a simple sign analysis, interpreting formulae from other modules using a compositional three-valued semantics.

refs (Section 4.7) provides simple tracking of heap references: each tracked variable is classified as either 'null', 'ref' (a valid heap reference) or 'other'.

HECTOR is written in Prolog, using SWI Prolog v5.6.55, and consists of about 17k lines of code. We chose to use Prolog due to a personal preference for the language, and because it has several features well-suited to rapid prototyping. Code can be developed in small pieces, which can then be easily tested in Prolog's interactive mode (a process greatly helped by the existence of little mutable state). Backtracking (non-determinism) can be used to quickly produce inefficient but nevertheless correct implementations of predicates. Lastly, Prolog's facilities for manipulating terms make representing and working with data such as $\mathscr{L}$-formulae very easy.

Figure 4.1 shows the overall structure of the implementation of HECTOR. This diagram can be regarded as a more precise, implementation-oriented version of our introductory Figure 1.1.

As a prototype, the HECTOR implementation is not highly "tuned" for speed, but we have introduced some simple optimisations which we detail. Where convenient, we exploited existing software; HECTOR invokes for various purposes TVLA [LAMS04, SRW99], Simplify [DNS05], dot [GN00], PiLLoW [HC01] and scheck [Lat03], as we will mention at appropriate points.

Between mid 2007 and early 2008 the web interface to HECTOR was publicly accessible. It is now our intention in the near future to release HECTOR under an open source license, once we have tidied up the code.

4.2 Implementation of module-based framework

Our implementation simply uses the Prolog database (via `assert` and `retract`) to store programs, configurations and models. A program is stored using the following collection of predicates:

```
:- dynamic field/1.              % field(FieldName).
:- dynamic proc/3.               % proc(ProcName, Params, LocalVars).
```

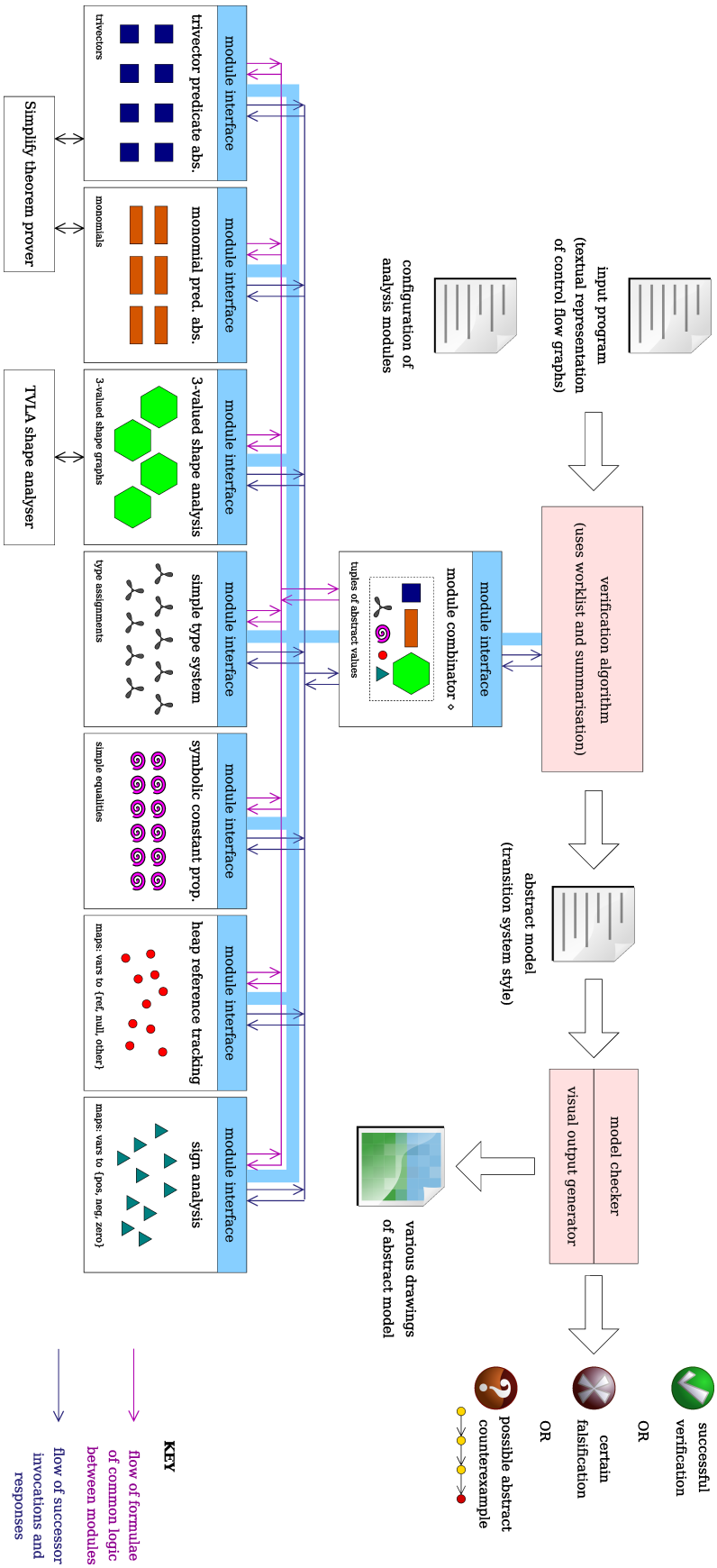

Figure 4.1: The overall structure of the HECTOR verification system. (This diagram can be regarded as a more precise, implementation-oriented version of our introductory Figure 1.1.)

```
:- dynamic edge/3.              % edge(ProcName, Location, Edge).
:- dynamic shorthand/2.         % shorthand(Name, Body).
```

HECTOR reads target programs in from text files. Figure 4.2 shows the file for our integer square root program from Chapters 2 and 3. Such files are in Prolog syntax already, so they can be loaded directly into the database. Note that formulae of $\mathscr{L}$ are simply represented by appropriate Prolog terms. The predicates used should be self-explanatory apart from perhaps `shorthand/2`, which allows one to declare shorthand names for large formulae, to make the program clearer. (`shorthand/2` also provides quick access to some common formula patterns; we lack the space to document this feature.) We emphasise that this shorthand mechanism is purely cosmetic and does *not* allow one to introduce inductively defined predicates.

Configuring the analysis is also done using a text input file (as shown in Figure 4.3); analysis modules are enabled and disabled, and configured on a per-procedure basis, with the following predicates:

```
:- dynamic enabled/1.           % enabled(ModuleName).
:- dynamic config/3.            % config(ProcedureName, ModuleName,
                                        Configuration).
```

Models are represented using a collection of Prolog predicates:

```
:- dynamic absnode/4.           % absnode(AbstractNodeID, ProcName,
                                          Location, AbstractValue)
:- dynamic edge/2.              %      edge(SourceNodeID, DestNodeID)
:- dynamic call_edge/2.         % call_edge(SourceNodeID, DestNodeID)
:- dynamic rtn_edge/2.          %  rtn_edge(SourceNodeID, DestNodeID)
```

With this scheme, loading and saving models is trivial — e.g. to load a model, only a single call to `load_files` is needed. It also means that the models are saved in a

```
shorthand(is_int_sqrt,
          and(
              lte(times(x, x), n),
              lt(n, times(plus(x,1), plus(x,1)))
             )
         ).


proc(main, [], [x, y]).

edge(main, start, call(x, chooseNat, [], 0)).
edge(main, 0, call(y, intSqrt, [x], terminated)).
edge(main, asserterror, skip(asserterror)).
edge(main, memerror, skip(memerror)).
edge(main, terminated, skip(terminated)).


proc(chooseNat, [], [n, one]).

edge(chooseNat, start, assignConst(one, 1, 0)).
edge(chooseNat, 0, choice(1, 2)).
edge(chooseNat, 1, arith(n, n, plus, one, 0)).
edge(chooseNat, 2, return(n)).
edge(chooseNat, asserterror, skip(asserterror)).
edge(chooseNat, memerror, skip(memerror)).


proc(intSqrt, [n], [x, one]).

edge(intSqrt, start, assignConst(one, 1, 0)).
edge(intSqrt, 0, if( lte(times(plus(x,1), plus(x,1)), n), 1, 2)).
edge(intSqrt, 1, arith(x, x, plus, one, 0)).
edge(intSqrt, 2, if(is_int_sqrt, 3, asserterror)).
edge(intSqrt, 3, return(x)).
edge(intSqrt, asserterror, skip(asserterror)).
edge(intSqrt, memerror, skip(memerror)).
```

Figure 4.2: An example target program in the form read in by HECTOR.

```
enabled(tpa).
enabled(compsigns).


config(chooseNat, compsigns, [n, one]).
config(main, compsigns, [x]).
config(intSqrt, compsigns, [n, x, one]).


config(intSqrt, tpa,
  [
    lte(times(x, x), n),
    withNeg(lte(times(plus(x,1), plus(x,1)), n)),
    eq(one, 1)
  ]).
```

Figure 4.3: An example configuration of the analysis, in the form read in by HECTOR.

human-readable and editable form, can be searched with ad-hoc Prolog queries, and can be manipulated with a host of other programs such as 'grep' and 'awk'. Storing data in text files like this is the UNIX tradition.

Using SWI Prolog's module system, we place the code for each *analysis* module in a separate *Prolog* module. Each such module provides predicates `init/1`, `share/4`, `succ/5`, `succ_c/5` and `succ_r/8` which correspond directly to the components of our interface for analysis modules (Definition 3.4.2). The $\diamond$ combinator (cooperative combination of analysis modules) is also implemented in such a Prolog module.

4.3 Implementation of sign analysis module (compsigns)

Our multi-variable sign analysis module is almost a direct translation of Subsection 3.4.2 into Prolog, and therefore we will not say much about it. To represent our finite partial functions we use SWI's built in `assoc` library. The only new point that arose during implementation was to exploit the "laziness" of some of the op-

erators involved, to avoid unnecessary computation. For example when abstractly evaluating $X + Y$, if X evaluates to 'any' there is no need to evaluate Y; the result will be 'any' in all cases. Similar cases exist for the Boolean connectives.

4.4 Implementation of predicate abstraction modules (tpa and mpa)

We have written modules which implement the trivector and monomial predicate abstraction techniques described in Subsection 2.5.1. We will describe the implementation of the trivector module only here, since the implementation of the monomial module is very similar and shares most of the code.

4.4.1 Connecting formulae

Typically, successor computation in a predicate abstraction system is done using a weakest precondition operator, or a strongest postcondition operator which we used when we introduced predicate abstraction in Subsection 2.5.1. Collectively these operators are called *predicate transformers*. However, we couldn't see how to use such operators in the presence of sharing; formulae received from other modules use two time indices, whereas those produced by the weakest precondition or strongest postcondition operators only use one. Thus we use what we call *connecting formulae* instead of a predicate transformer.

The connecting formula for a statement s is so-called because it connects the state before the execution of s to the state afterwards. In our example program from Figure 3.1, the procedure 'chooseNat' contains a statement $\texttt{Arith}(n, n, +, one)$, i.e. `n := n + one`. The connecting formula for this statement is:

$$n = n_{\mathbf{I}} + one_{\mathbf{I}} \quad \wedge \quad one = one_{\mathbf{I}} \quad \wedge \quad \forall X(allocd(X) \leftrightarrow allocd_{\mathbf{I}}(X))$$

The conjunct which asserts that the allocation set is not changed by the statement uses a quantifier; if the program included any heap fields, each field would similarly introduce a universal quantifier.

There are also connecting formulae for procedure calls and returns. Figure 4.4 shows the interpretation of the time indices $\mathfrak{0}$, $\mathfrak{I}$ and $\mathfrak{C}$ when procedure calls are processed. For the call to 'intSqrt' in the 'main' procedure, the connecting formula is

$$one = 0 \quad \wedge \quad x = 0 \quad \wedge \quad n = x_{\mathfrak{I}} \quad \wedge \quad \forall X(allocd(X) \leftrightarrow allocd_{\mathfrak{I}}(X))$$

(Because of the time indices, there is no confusion between the variable x in the 'main' procedure, and the variable with the same name in the 'intSqrt' procedure.)

Returns have the most complicated connecting formulae. Figure 4.5 shows the interpretation of the time indices for returns; this is where the time indices $\mathfrak{2}$ and $\mathfrak{Z}$ come into play. For the return from 'intSqrt' back to 'main', the formula is:

$$x_{\mathfrak{I}} = x \quad \wedge \quad y = x_{\mathfrak{Z}} \quad \wedge \quad n_{\mathfrak{2}} = x_{\mathfrak{I}}$$
$$\wedge \quad \forall X(allocd_{\mathfrak{I}}(X) \leftrightarrow allocd_{\mathfrak{2}}(X)) \quad \wedge \quad \forall X(allocd_{\mathfrak{Z}}(X) \leftrightarrow allocd(X))$$

This formula encodes the returning of the result, but also the passing of parameters, *even though this has been done once already* in the connecting formula for the call. This is crucial because otherwise frame conditions established in the called procedure cannot be properly used in the analysis of the calling procedure.

Finally there is also a connecting formula for initialisation, which states that the local variables of the main procedure π_1 begin with the value 0; for our example program this is $x = 0 \wedge y = 0$.

We wonder how much predicate abstraction suffers from using connecting formulae rather than a predicate transformer. Using connecting formulae seems to result in larger formulae being sent to the theorem prover, yet formula size is not necessarily

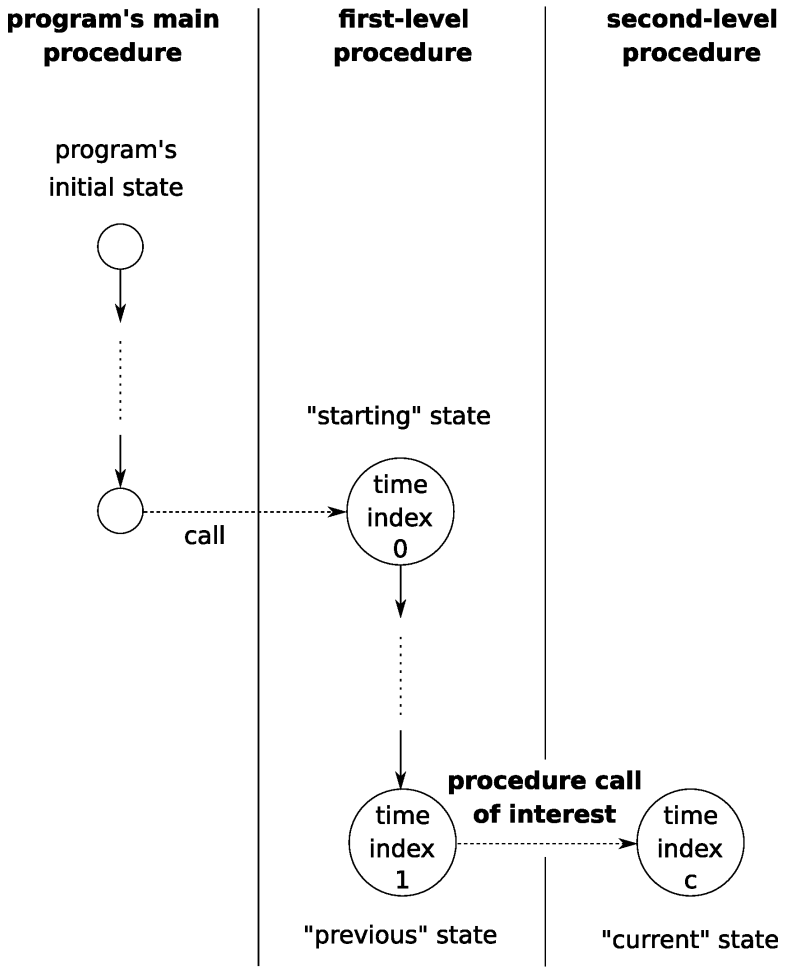

Figure 4.4: An illustration of the roles of the time indices $\mathfrak{O}$, $\mathfrak{I}$ and $\mathfrak{C}$ when considering a procedure call.

a good measure of how "difficult" a formula is for the prover. Our suspicion is that the equalities between different "versions" of the same variable, e.g. $x = x_{\mathfrak{I}}$, are well-handled by a Nelson-Oppen style prover like Simplify, but that such a prover may have more trouble with the conjuncts which express the non-change of allocation sets and fields, because these involve quantifiers.

4.4.2 Formulating predicate abstraction as a module

Now that we have our connecting formulae, it is fairly easy to implement the trivector predicate abstraction technique as an analysis module $\mathbf{tpa}\,\langle \rho, \Delta \rangle$. Here the module is parametrised by the theorem prover ρ to be used, and the selection of abstraction

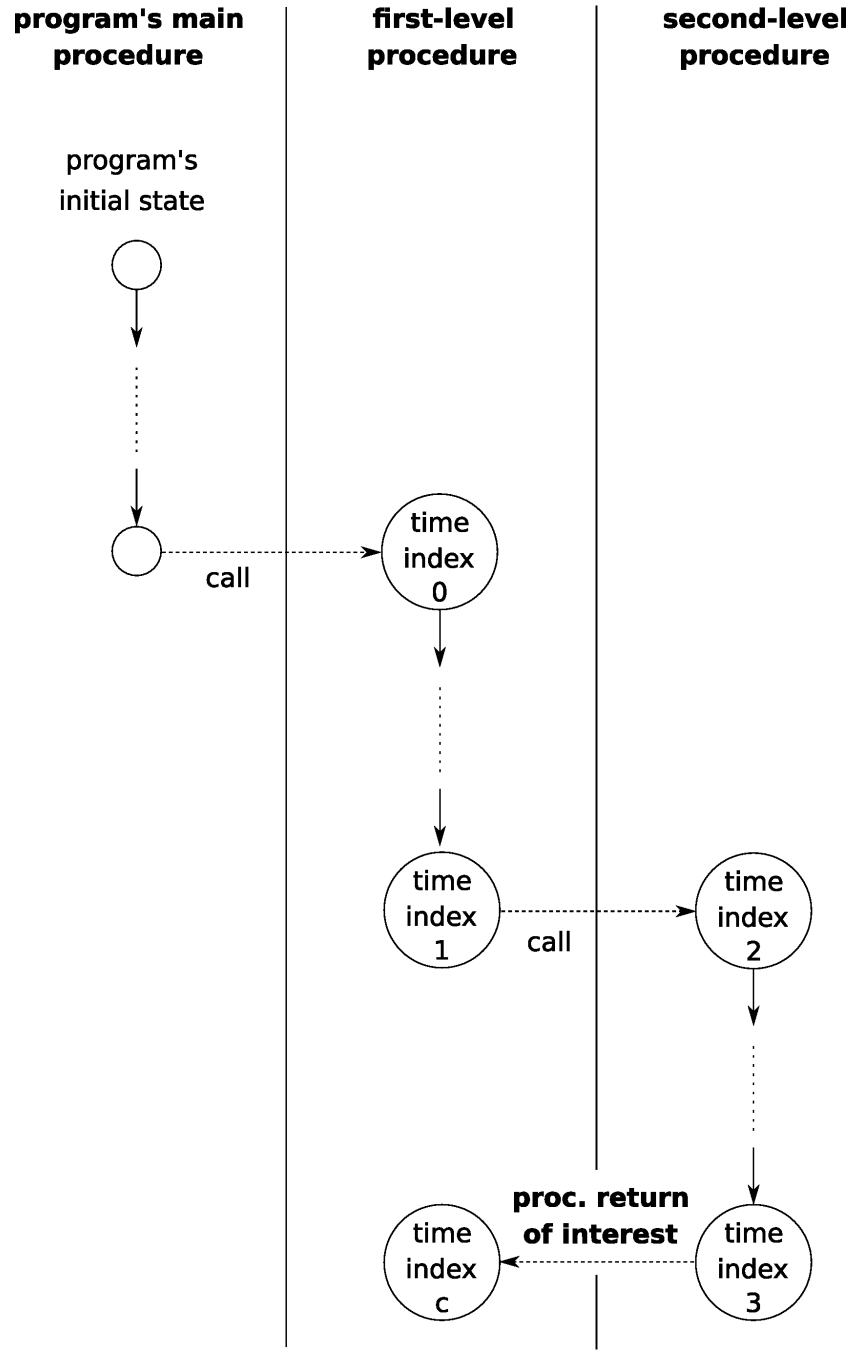

Figure 4.5: An illustration of the roles of the five time indices $\mathfrak{0}$, $\mathfrak{1}$, $\mathfrak{2}$, $\mathfrak{3}$ and $\mathfrak{C}$ when considering a procedure return.

predicates: Δ maps each procedure in the program to a list

$$\Delta(\pi) = [P^1, \ldots, P^n]$$

of abstraction predicates to be used in that procedure, where each P^i is in $\mathscr{L}^{\{\mathfrak{O},\mathfrak{C}\}}$. Then T contains conjunctions of these formulae or their negations.

$$T \quad := \quad \left\{ \Psi^1 \wedge \cdots \wedge \Psi^k \;\middle|\; \begin{array}{r} \text{each } \Psi^i \text{ is from } \{P^i, \neg P^i, \mathit{true}\} \\ \text{where } [P^1, \ldots, P^n] = \Delta(\pi) \text{ for some } \pi \in \mathit{Procs}(P) \end{array} \right\}$$

Concretisation is easy: since the abstract values are formulae of $\mathscr{L}$, describing the starting state and the current state, we map each to the set of pairs of states where it is true:

$$\gamma(\Psi) \quad := \quad [\![\Psi]\!]^{\{\mathfrak{O},\mathfrak{C}\}}$$

Our definition of *share* is likewise simple. We make *share* propagate the entire abstract value[1], which is already a formula (substituting time index $\mathfrak{I}$ for $\mathfrak{C}$, to indicate that we are describing the state before execution of the statement of interest):

$$\mathit{share}(\Psi, s, \pi) \quad := \quad \Psi[\mathfrak{C} \backslash \mathfrak{I}]$$

Successor computation follows the same pattern as in Example 2.5.2, i.e. we construct a formula Θ expressing "what we know" about the new state, and then test each abstraction predicate P^i in turn, to see whether either P^i or $\neg P^i$ follow from Θ. The difference is that instead of being constructed with the strongest postcondition operator *SP*, Θ is now a conjunction of

[1] We haven't made *share* also propagate the connecting formula. This would be sound, but seems pointless because the connecting formula only expresses the semantics of the language, which all the modules should know about anyway. Having said that, early in the work we had a separate "semantics" module whose sole job was to share the connecting formulae; the predicate abstraction modules then required no notion of the semantics of the language. This gives rise to interesting possibilities for also making the system parametric w.r.t. the programming language used, but had to be abandoned when we decided not to use formula sharing during procedure calls and returns.

- the (time-substituted) abstract constraint $(\hat{\Psi}^1 \wedge \cdots \wedge \hat{\Psi}^n)[\mathfrak{O}\backslash\mathfrak{I}]$ whose successor we are computing,
- the connecting formula, and
- the extra constraints Φ received from other modules (this is how we take advantage of shared information).

Thus a first attempt at defining the *succ* function is

$$succ'(\hat{\Psi}^1 \wedge \cdots \wedge \hat{\Psi}^n, s, \pi, \Phi) \quad := \quad \{\Psi^1 \wedge \cdots \wedge \Psi^n\}$$

where

$$\Theta \quad := \quad (\hat{\Psi}^1 \wedge \cdots \wedge \hat{\Psi}^n)[\mathfrak{O}\backslash\mathfrak{I}] \quad \wedge \quad connect(s, \pi) \quad \wedge \quad \Phi$$

and

$$[P^1, \ldots P^n] = \Delta(\pi)$$

and each Ψ^i is given by

$$\Psi^i \quad := \quad \begin{cases} P^i & \text{if } \rho \text{ proves } \Theta \rightarrow P^i \\ \neg P^i & \text{if } \rho \text{ proves } \Theta \rightarrow \neg P^i \\ true & \text{otherwise} \end{cases}$$

This is sound but, just as with the sign analysis module **compsigns** , the information received from other modules may actually contradict the abstract value, so it's a good idea to first check whether the known information Θ is consistent, and generate an empty set of successors if it is not. Thus we define

$$succ(\hat{\Psi}^1 \wedge \cdots \wedge \hat{\Psi}^n, s, \pi, \Phi) \quad := \quad \begin{cases} \emptyset & \text{if } \rho \text{ proves } \neg\Theta \\ succ'(\hat{\Psi}^1 \wedge \cdots \wedge \hat{\Psi}^n, s, \pi, \Phi) & \text{otherwise} \end{cases}$$

where Θ is as before. The above definitions implicitly account in a sound way for the undecidability of the logic $\mathscr{L}$. The functions $succ_C$, $succ_R$ and *init* are implemented similarly.

Remark: consistency checking We can obtain the consistency checking module **cons** of Subsection 3.6.2 from either of the predicate abstraction modules, by choosing an empty list of abstraction predicates for all procedures, and replacing the connecting formulae with *true*.

4.4.3 Interfacing with the theorem prover

We have used the existing theorem prover Simplify [DNS05]. We give Simplify a few basic axioms about our allocation predicates, e.g.

$$\forall X \, allocd(X) \rightarrow X > 0$$

and some axioms concerning integer multiplication. Simplify has no built-in support for transitive closure, so TC subformulae are systematically replaced with uninterpreted predicates applied to the appropriate terms.

Our initial intention was not to give Simplify any axioms for reasoning about TC formulae (such as those in Figure 2.5.3). We believed that TC reasoning could be left entirely to the shape analysis module. However in practice we discovered one specific situation in which this plan fails: because we have implemented sharing only for intraprocedural statements, at procedure calls and returns the shape analysis module is unable to help out with the TC reasoning. Fortunately, since the heap doesn't change during parameter passing and value return, a very limited form of TC reasoning suffices. We won't give the full details, but when we see a certain kind of subformula of the form

$$TC_{[A,B]} \left[\Phi(A, B)\right] (t, t')$$

we generate axioms of the form

$$\forall X, Y \left(\Phi(X,Y) \leftrightarrow \Phi[\mathfrak{j}\backslash\mathfrak{k}](X,Y) \right)$$

$$\rightarrow$$

$$\forall X, Y \left(TC_{[A,B]} \left[\Phi(A,B)\right](X,Y) \quad \leftrightarrow \quad TC_{[A,B]} \left[\Phi[\mathfrak{j}\backslash\mathfrak{k}](A,B)\right](X,Y) \right)$$

where $\mathfrak{j}, \mathfrak{k} \in \mathit{Time}$ are distinct time indices (recall that $[\mathfrak{j}\backslash\mathfrak{k}]$ denotes "time substitution"). Each such axiom is an instantiation of the idea that if two relations are equal, then their transitive closures must be equal also, and allows TC properties to be "carried over" from one time index to the next, provided the underlying relation has not changed. These axioms could be dropped if we implemented formula sharing for calls and returns.

We chose the Simplify prover because it has been used in many program verification systems (a partial list is included in [DNS05]). However, since the theorem prover is hidden behind a well-defined interface, another prover could be used instead with minimal changes to the HECTOR code.

4.5 Implementation of shape analysis module (tvla)

In this section we describe the shape analysis module we have written. Of the modules currently included in HECTOR, the shape analysis module, **tvla** , has by far the most difficult and involved implementation, and consequently this section is quite long.

4.5.1 Basic implementation and choice of core predicates

Our shape analysis module is built using (a slightly modified version of) the existing TVLA software. As described in Subsection 2.5.3 (page 65 onwards) TVLA uses

three-valued heap models, such as the one in Figure 2.10, to represent sets of concrete heaps. Thus the abstract value space T is the set of such three-valued heaps, along with additional elements *givenup* and *ignoring*, whose roles will be explained shortly.

We take the domain of our TVLA models to be the set of allocated heap objects rather than the set of all heap addresses. We need to track the heap over the three time indices $\mathfrak{0}$, $\mathfrak{1}$ and $\mathfrak{C}$, so all of our predicates come in three versions ([JLRS04] takes a similar approach, but only uses two states instead of three; in the language of that paper, our vocabulary is tripled instead of doubled). As usual with TVLA, we represent our heaps with concrete predicates of two kinds:

predicates for variables: For each variable v in a procedure π, for each time index $\mathfrak{j}$, there is a unary predicate named `pi_C_Var[v]_j` which indicates where the variable points.

predicates for fields: For each field f in the program, for each time index $\mathfrak{j}$, there is a binary predicate named `C_Field[f]_j` which indicates where the field points.

The declarations of these core predicates, and their update rules for each statement, are generated from the target program by our module.

As well as being the most complicated module we implemented, our shape analysis module is the most configurable. The following aspects of its behaviour can be "tuned":

Tracked variables and ignored variables The module can be instructed not to model certain program variables. Such variables we call *ignored variables*, and the remaining variables we call *tracked variables*. A common reason to omit a variable from the shape analysis is that it is used to store integer data values rather than object addresses. Ignoring irrelevant variables gives rise to

smaller abstract heaps, but also *fewer* of them, because heaps are no longer considered unequal solely because of different values of the irrelevant variables.

Tracked fields and ignored fields The comments about tracked versus ignored variables also apply to fields.

Tracked procedures and ignored procedures The module can also be instructed not to perform shape analysis at all in certain procedures, which we call *ignored procedures*. During the analysis of such procedures, the special value *ignoring* will be used as this module's abstract value. A call to an ignored procedure is handled by taking the abstract heap at the call point and copying it unchanged to the return point. Hence, a procedure can only be soundly ignored if it doesn't update any tracked fields, doesn't allocate any new heap objects, and doesn't invoke any tracked procedures. (This condition is enforced by our module.)

Instrumentation for reachability Currently our module can generate the declarations and update rules for two kinds of reachability instrumentation:

- A predicate named `pi_I_reach_v_f_j` can be generated to track whether each heap object is reachable from the program variable v in procedure π by following the field f (at time index $\mathfrak{j}$).
- A predicate named `I_cycle_f_j` can be generated to track whether each heap object lies on a cycle of pointers of the f field (at time index $\mathfrak{j}$).

Part of the module's configuration consists of choosing for which variables and fields these instrumentation predicates should be generated.

Focusing The *focus* operation [SRW99] is intuitively a kind of case-splitting: when a formula evaluates in a particular heap to the third truth value *unknown*, focusing splits into two cases, one where the formula is true, and another where it is false. This can lead to improved precision.

Our module performs focusing only on the `pi_C_Var[v]` predicates which represent (current) program variables. Part of the module's configuration consists of choosing for which variables in which procedures focusing should be performed.

Sharing patterns The final part of the configuration is to decide what kinds of properties the shape analysis module should share with other modules. This is done by specifying patterns of formulae that should be considered for sharing, as we will detail in Subsection 4.5.5.

4.5.2 Cases where shape analysis "gives up"

A significant design decision taken in our module is that we consider a variable v to be null (in procedure π at time index $\mathfrak{j}$) if the corresponding predicate `pi_C_Var[v]_j` is false everywhere, i.e. there is no object to which the variable may point. We make the same interpretation for fields.

This choice simplifies the representation of heaps, because there is no need to maintain separate information about the nullness of variables and fields; but, problematically, it also leaves us with no way to represent the situation where a tracked variable is neither null nor pointing to any object (e.g. a variable used for integer data).

Our solution is to identify statements which might introduce such a situation, and treat them with special care. Such statements include for instance assigning to a tracked variable from an ignored variable, reading into a tracked variable from an ignored field, and assigning the result of an arithmetic operation to a tracked variable. However, it suffices to correctly handle the copying of values from ignored variables to tracked ones, because other problematic statements can be transformed to first put their result into a new temporary ignored variable. When computing successors for such a statement $\texttt{VarCopy}(u, v)$, where u is tracked and v is ignored,

the *succ* function first inspects the shared information received. If this contains as a conjunct one of the formulae

$$v_{\mathfrak{I}} = 0, \qquad allocd_{\mathfrak{I}}(v_{\mathfrak{I}}), \qquad v_{\mathfrak{I}} = 0 \vee allocd_{\mathfrak{I}}(v_{\mathfrak{I}})$$

then the analysis can continue normally. Otherwise, the shape analyser "gives up", returning as sole successor the special value *givenup*. In this way, the shape analysis module uses information from other modules to overcome its own shortcomings. The successor of *givenup* under any operation is also *givenup* so, in that particular branch of the model, no further shape analysis will be attempted.

4.5.3 Treatment of procedure calls and returns

Our treatment of procedure calls and returns (for tracked procedures) is similar to that in [JLRS04], in that we use a multiple vocabulary (i.e. doubled or tripled) to discover relations between states at the beginning and end of a called procedure. When we compute the abstract state upon entry to a called procedure, the TVLA operations we invoke to do this will automatically discard details of the parts of the heap which are not of interest to the called procedure. This is also similar to [JLRS04], and is good, because it gives smaller abstract heaps and makes procedure summarisation perform better.

The downside of this arrangement, however, becomes apparent when it is time to process the corresponding procedure return: any heap structure not relevant to the called procedure has been lost. How can one recover it? For a subclass of programs called "cutpoint-free programs", one approach given in [RSY05] performs *local* reasoning about procedures, which involves partitioning the heap at the call site into the part accessed by the procedure, and the rest of the heap (which will not be changed in the call). This is very reminiscent of separation logic.

We considered implementing this "local reasoning" approach but found that, be-

cause our HECTOR system allows sharing of information between modules, a more lightweight solution is possible, which we now describe. *Within our shape analysis module* we simply "bite the bullet" and accept that after a procedure has returned a lot of information about the heap will be lost. However, there may be other modules in the system, such as the predicate abstraction modules, which, using frame conditions from the called procedure, still maintain constraints about the overall heap structure. Using these constraints and the sharing mechanism, this information can then, to an extent, be "put back" into the shape analysis.

4.5.4 Using shared formulae: translation followed by coercion

In this subsection we describe how our shape analysis module uses shared formulae in the common logic $\mathscr{L}$ to make its abstract heaps more precise. (This is in addition to the special use, previously described, of shared formulae for deciding whether to "give up" or not). There are two stages to the process: firstly, the formulae of $\mathscr{L}$ are translated into the logic used internally by TVLA, and secondly the translated formulae are applied to the abstract heap using the *coercion* operation. Coercion can result in the detection of inconsistency between the abstract heap and the shared formulae, but also, more subtly, in the "sharpening" of the abstract heap by replacing some *unknown* values with *true* or *false*.

Translation of formulae Since TVLA's internal logic is also a first order logic with transitive closure, at first glance it seems that formulae of $\mathscr{L}$ are in the right form already. Three difficulties present themselves, however.

1. The universes over which the logics are interpreted are different. In $\mathscr{L}$ variables range over $\mathbb{Z}$, whereas in TVLA's logic they range over (just) the allocated objects.

Rules for Boolean connectives:

$$\begin{aligned}
true^\dagger &\triangleq true \\
(\neg\Phi)^\dagger &\triangleq \neg(\Phi^\dagger) \\
(\Phi_1 \wedge \Phi_2)^\dagger &\triangleq (\Phi_1^\dagger) \wedge (\Phi_2^\dagger) \\
(\Phi_1 \vee \Phi_2)^\dagger &\triangleq (\Phi_1^\dagger) \vee (\Phi_2^\dagger) \\
(\Phi_1 \rightarrow \Phi_2)^\dagger &\triangleq (\Phi_1^\dagger) \rightarrow (\Phi_2^\dagger) \\
(\Phi_1 \leftrightarrow \Phi_2)^\dagger &\triangleq (\Phi_1^\dagger) \leftrightarrow (\Phi_2^\dagger)
\end{aligned}$$

Rules for quantifiers:

$$\begin{aligned}
(\exists X(\mathit{allocd}(X) \wedge \Phi))^\dagger &\triangleq \exists X(\Phi^\dagger) \\
(\exists X(\mathit{allocd}_1(X) \wedge \Phi))^\dagger &\triangleq \exists X(\neg \mathit{isNew}(X) \wedge (\Phi^\dagger)) \\
(\forall X(\mathit{allocd}(X) \rightarrow \Phi))^\dagger &\triangleq \forall X(\Phi^\dagger) \\
(\forall X(\mathit{allocd}_1(X) \rightarrow \Phi))^\dagger &\triangleq \forall X(\neg \mathit{isNew}(X) \rightarrow (\Phi^\dagger))
\end{aligned}$$

Figure 4.6: Our translation $\Phi \mapsto \Phi^\dagger$ from $\mathscr{L}$ to TVLA's internal logic. (The predicate *isNew* is built into TVLA and identifies new objects allocated by the current statement.)

2. In $\mathscr{L}$, fields are encoded with functions and program variables are encoded with variables; but with TVLA relations are used for both.

3. It is not enough for the translation to be correct in the two-valued sense; we must also consider the three-valued behaviour of the translated formula. This is because even when two formulae are equivalent over two-valued models, one may be more precise over three-valued models. (See Appendix A.1 for an example of this phenomenon.)

Our translation, $\Phi \mapsto \Phi^\dagger$, handles Boolean connectives and quantifiers in a structural manner, as given by the rules in Figure 4.6. Note that we only translate quantifiers whose variables are suitably "guarded" by an allocation predicate; for these formulae the problem of differing universes goes away.

Translation of literals and transitive closure subformulae How are we to translate the literals? An early version of our translation, included in a technical report [Cha06b], treated the literals very systematically. That version included a scheme for translating arbitrary $\mathscr{L}$ terms, and then gave translated versions of the relation symbols such as $=$ and $\neq$.

Unfortunately, our systematic translation did not work well. The problem, we discovered, was that it produced results that were correct in the two-valued sense, but performed poorly under three-valued evaluation. Though we were well aware of this phenomenon in theory, its occurrence in practice surprised us repeatedly. For example, translation of an equality between a logical variable and some other term could give rise to a subformula of the shape

$$\exists X(X = u \wedge \Phi(X))$$

which is less precise under three-valued evaluation than $\Phi(u)$, since if u and X are bound to the same summary node the equality $X = u$ will only evaluate to *unknown*.

Instead, then, our translation of literals uses a large number of cases; for instance, instead of a single rule for forming $(t = t')^\dagger$, we have a number of rules depending on the specific forms of the terms t and t'. For example, if x and y are program variables in the procedure π, and f is a field, then $(x = y)^\dagger$ is

$$\forall o(\texttt{pi\_C\_Var[x]}(o) \leftrightarrow \texttt{pi\_C\_Var[y]}(o))$$

and $(f(x) = 0)^\dagger$ is

$$\begin{array}{rl} & \Big(\exists o(\texttt{pi\_C\_Var[x]}(o) \wedge \neg\exists o'\texttt{C\_Field[f]}(o,o'))\Big) \\ \vee & \Big((\forall o(\texttt{pi\_C\_Var[x]}(o) \rightarrow \neg\exists o'\texttt{C\_Field[f]}(o,o'))) \quad \wedge \quad \mathit{unknown}\Big) \end{array}$$

In the second of these, we use *unknown* because we don't want to say anything about

what happens when x is null. Frequently, as in this case, the translation has the structure

$$\text{(sufficient condition)} \vee (\text{necessary condition} \wedge \textit{unknown})$$

which allows both falsifying and establishing the condition.

While it is irritating to have to implement such cases for literals by hand, we found that it does give much better results. Transitive closure subformulae are treated in a similar way. Any subformula for which there is no translation rule, such as one formed from a quantifier that is not appropriately "guarded" by an allocation predicate, is simply translated to *unknown*; while imprecise, this is always sound.

Coercion Once we have obtained by translation a formula in TVLA's internal logic, it is applied to the abstract heap using the *coercion* operation detailed in [SRW99]. The formula is transformed into a set of *coercion rules*, each of the form

$$\textit{Body} \rhd \textit{Head}$$

where *Body* can be any formula but *Head* can only be an atomic formula or the negation of an atomic formula. Free variables are taken as universally quantified. If *Body* evaluates to *true* in the abstract heap, but *Head* evaluates to *false*, the heap can be thrown away. Alternatively, if *Body* evaluates to *true* and *Head* to *unknown*, the heap is altered to *force* the *Head* part to be true, by "sharpening" some predicates from *unknown* to a definite value.

As in other applications of TVLA, our module also uses a set of coercion rules to encode some basic facts about heaps. For instance the rule

$$p \neq q \wedge \texttt{pi\_C\_Var[x]}(p) \quad \rhd \quad \neg\texttt{pi\_C\_Var[x]}(q)$$

reflects that fact that a program variable cannot simultaneously point to two distinct

procedure SHARE(a, s, π)

var $F : \mathscr{L}^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$ set
var $\Phi : \mathscr{L}^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

$F :=$ CANDIDATES(s, π)
$\Phi := \mathit{true}$

while $F \neq \emptyset$ **do**

choose $\Psi \in F$
$F := F - \{\Psi\}$

if SUCC(a, s, π, $\neg\Psi$) $= \emptyset$ **then**
$\Phi := \Phi \wedge \Psi$
end if

end while

return Φ

end procedure

Figure 4.7: By exploiting $\Phi \mapsto \Phi^{\dagger}$, our translation *to* TVLA's internal logic from $\mathscr{L}$, we can do the reverse, i.e. extract information *from* TVLA back into the common logic $\mathscr{L}$. The procedure CANDIDATES suggests candidate formulae for sharing. This scheme will also work for any similar translation.

objects.

4.5.5 Providing shared formulae

In this subsection we explain a method for extracting a set of $\mathscr{L}$-formulae which are entailed by a given abstract heap. These formulae can then be passed to other modules via the *share* function.

Existing work such as [Yor03] could be exploited for extracting first-order formulae directly from abstract heaps. But because our HECTOR system is a prototype, we instead apply a cheap trick which reuses the translation from $\mathscr{L}$ into TVLA to induce transfer of information in the other direction. This quickly gives us a usable implementation with little work.

Our method is given in Figure 4.7. The idea is to generate, independently of the abstract heap, a set of *candidate* formulae which we consider propagating to the other modules. Then, to test whether a candidate formula Ψ is really entailed by the abstract heap, we run *succ*, putting in the negation of Ψ. If *succ* returns no successors, this indicates that $\neg\Psi$ contradicted the abstract heap, and therefore Ψ is entailed by the abstract heap. We gather up all candidates which are shown to be entailed in this way, and return their conjunction.

This still leaves the question of how one obtains the candidates. We mentioned earlier that the candidates are generated from patterns given as part of the configuration. Without going into detail, we mention that such patterns are specified using a small subset of Prolog, so that for instance the pattern

$$\texttt{neq}(\texttt{U},\texttt{V}) : \texttt{var}(\texttt{U}), \texttt{var}(\texttt{V}), \texttt{U} < \texttt{V}$$

generates, without unnecessary repetition, all potential disequalities between program variables. Of course "singleton" patterns, which directly and fully specify a formula, can be used in the configuration if it is known exactly what should be considered for sharing.

It is worth pointing out that the method we use here, and the machinery we created for specifying patterns of candidate formulae, can be applied unchanged to any other analysis module.

4.6 Implementation of a simple type system module (types)

4.6.1 A simple type system for heap references

We begin by presenting a simple type system which identifies variables which hold heap references. Although our variables and fields all contain integer values, we can broadly classify them according to their use: some are used to store integer data values, and some are used to store the addresses of heap objects. We have designed a simple type system which tries to distinguish these two uses. This system classifies each variable and field as having either the type *Any* (for data) or the type *Ref* (for heap addresses). Variables/fields of type *Any* can hold any value, whereas variables/fields of type *Ref* must contain either the address of an allocated heap object, or 0 i.e. null. Variables which store addresses calculated by pointer arithmetic must be conservatively typed as *Any*; values can flow from *Ref* to *Any* but not vice versa.

We will need a notion of a *type assignment* Γ. Conventionally Γ would be formalised as a function from variables/fields to types, but because our type system is so simple, we can take a type Γ to be the set of variables and fields which are given the *Ref* type.

Definition 4.6.1. A *type assignment* Γ is a subset

$$\Gamma \quad \subseteq \quad \mathit{Fields} \cup (\mathit{ProcNames} \times \mathit{Vars})$$

A program P is well-typed by a type assignment Γ if the following conditions hold, for every statement s which labels an edge in a procedure $\pi \in \mathit{Procs}(P)$:

- If s has the form $\texttt{VarCopy}(u, v) : n$ then $(\pi, u) \in \Gamma \implies (\pi, v) \in \Gamma$.
- If s has the form $\texttt{AssignConst}(u, k) : n$ then either $k = 0$, or $(\pi, u) \notin \Gamma$.

- If s has the form $\texttt{Arith}(u, v_1, \otimes, v_2) : n$ then $u \notin \Gamma$.
- If s has the form $\texttt{FieldRead}(u, v, f) : n$ then $(\pi, u) \in \Gamma \implies f \in \Gamma$.
- If s has the form $\texttt{FieldWrite}(v, f, u) : n$ then $f \in \Gamma \implies (\pi, u) \in \Gamma$.
- If s has the form $\texttt{Call}(u, \pi', [a_1, \ldots, a_k]) : n$ then, letting $[p_1, \ldots, p_k]$ be the formal parameters $\mathit{Formals}(\pi')$, for $i = 0, \ldots, k$ we have $(\pi', p_i) \in \Gamma \implies (\pi, a_i) \in \Gamma$. Also, letting r be the variable returned in π', we have $(\pi, u) \in \Gamma \implies (\pi', r) \in \Gamma$.

Informally, these conditions prevent any opportunity for values to flow from a variable/field of type *Any* into one of type *Ref*. □

The following theorem concerns the behaviour of well-typed programs; it guarantees that at all stages of execution, variables which are typed *Ref* really do contain either 0 or an allocated heap address, and similarly for fields.

Theorem 4.6.2. Let the program P be well-typed by the type assignment Γ, and let execution in P reach a state $s = (e, h, A)$ in procedure π, i.e. $\pi, s_0 \xrightarrow{p} l, s$. Then:

1. For each variable v in procedure π (i.e. $v \in \mathit{Formals}(\pi)$ or $v \in \mathit{Locals}(\pi)$), if $(\pi, v) \in \Gamma$ (i.e. v has type *Ref*) then either $e(v) = 0$ or $e(v) \in A$.
2. For each field $f \in \mathit{Fields}(P)$, if f has type *Ref* (i.e. if $f \in \Gamma$) then, for all $a \in A$, either $h(f, a) = 0$ or $h(f, a) \in A$.

Proof: The proof is straightforward and is omitted for brevity. Proceed by induction on the number of steps in the derivation of $\pi, s_0 \xrightarrow{p} l, s$. □

4.6.2 Turning our type system into an analysis module

We can now turn this type system into a module **types** which can be used in our cooperative analysis. When the module is first loaded, it runs a simple type inference

scheme over the entire target program, to find a type assignment Γ_P that well-types the program P. Thereafter, apart from *share*, its functions have nothing to do:

$$\begin{aligned}
T &:= \{\mathrm{Um}\} \\
\gamma(\mathrm{Um}) &:= \mathit{State} \times \mathit{State} \\
\mathit{succ}(a, s, \pi, \Phi) &:= \{\mathrm{Um}\} \\
\mathit{succ}_C(a, \pi, \pi', [a_1, \ldots, a_k]) &:= \{\mathrm{Um}\} \\
\mathit{succ}_R(a, a', \pi, \pi', x, r, [a_1, \ldots, a_k]) &:= \{\mathrm{Um}\} \\
\mathit{init}(\cdot) &:= \{\mathrm{Um}\}
\end{aligned}$$

When invoked, *share* looks up the in-scope variables in the pre-computed type assignment Γ_P and supplies a formula which appropriately constrains each *Ref*-type variable or field. Specifically, for each procedure $\pi \in \mathit{Procs}(P)$ we define a formula constraining its *Ref*-type variables

$$\Phi(\pi) \quad := \quad \bigwedge_{(\pi,v)\in\Gamma} (v_{\mathbf{I}} = 0 \vee \mathit{allocd}_{\mathbf{I}}(v_{\mathbf{I}}))$$

plus a formula to constrain the fields:

$$\Phi(\mathit{fields}) \quad := \quad \bigwedge_{f\in\Gamma} \forall X (\mathit{allocd}_{\mathbf{I}}(X) \rightarrow (f_{\mathbf{I}}(X) = 0 \vee \mathit{allocd}_{\mathbf{I}}(f_{\mathbf{I}}(X))))$$

Then *share* is defined by

$$\mathit{share}(a, s, \pi) \quad := \quad \Phi(\pi) \wedge \Phi(\mathit{fields})$$

The reader may wonder why this module is sound — the information it shares certainly isn't entailed by the abstract value, as there is only one, namely 'Um', and it conveys nothing. The reason is the "non-locality" in the soundness conditions, which we noted in Subsection 3.4.3: in each of the soundness conditions in Defini-

tion 3.4.3, there is a premise $\pi, s_0 \xrightarrow{l,s} l', s'$, which ensures that the concrete state being considered is actually reachable. This is precisely the condition needed in Theorem 4.6.2 to obtain the result we need.

4.6.3 Additional types and type inference

The **types** module we implemented features additional types: *bool* for Boolean variables (which take value 0 or 1), *nneg* for non-negative integers, and *null* for variables which never change their value from the initial null.

types uses a type inference algorithm to automatically discover the best types for variable and fields. Type inference was implemented in a relatively straightforward way using SWI Prolog's constraint programming features.

Remarks Everything provided by the above type module can also be obtained using predicate abstraction with suitable predicates. However, the type system module works more efficiently: it performs a cheap static program-wide check once, rather than causing the theorem prover to do extra work at every iteration.

Our module **types** is fairly simple, though nevertheless it is useful (as we see in Section 5.4). It also serves to demonstrate how type systems can fit into our framework. It would be interesting to add non-null reference types as in the Cyclone language [JMG+02]. Another possibility is to add one of the ownership type systems referred to in Subsection 2.5.3 (from page 68).

4.7 Implementation of two further shallow domains

4.7.1 Symbolic constant propagation (symbprop)

The fact that our target language has such a small set of statement forms tends to hamper the use of predicate abstraction. Suppose we are using a single abstraction predicate $P_1 \triangleq x = a+b+c$. After the execution of the statement `x := a + b + c` the analysis will establish P_1. But if we implement the same operation in our minimal language, using two statements `x := a + b; x := x + c`, then afterwards P_1 will not be established; the reason is that the information that $x = a + b$ is lost between the statements. To overcome this, an extra abstraction predicate $P_2 \triangleq x = a + b$ must be introduced. In general, when using the minimal language a great many more abstraction predicates are required. Other domains become similarly impeded.

Symbolic constant propagation [Min06, LF08] was designed to combat this phenomenon, by propagating obvious information, such as the $x = a + b$ above, a short distance forward in the analysis. We will not describe the details of our module **symbprop**, but note that in the above example, it generates the constraint $x = a + b$ and carries it through the analysis until one of the variables involved is modified. Guards of `If` edges are carried forward in a similar way.

symbprop is very cheap because it never invokes a theorem prover; constraints are abandoned as soon as there is a modification to one of the variables or fields involved. (By "modification" we mean the variable or field appears on the left of an assignment. We do not care whether the value *actually changes*; it is enough that there is syntactically a potential change.)

4.7.2 Tracking of heap references (refs)

Our **refs** module provides simple tracking of heap references. Each tracked variable is classified as either 'null', 'ref' (a valid heap reference) or 'other'. This domain is an invention of our own, but there is nothing cunning about it. The implementation follows the same pattern as that of sign analysis, including the three-valued interpretation of formulae received from other modules.

4.8 Optimisations

As stated earlier, we have not made a serious attempt to make HECTOR run quickly. Nevertheless, we have made a few obvious optimisations, which we now briefly describe.

Indexing of formulae and caching of theorem prover results Inside HECTOR, formulae of the common logic $\mathscr{L}$ are passed from one piece of Prolog code to another very frequently, and some of these formulae become very large. Hence, rather than pass around the formulae themselves, we store all the "active" formulae in a table, and use their indices as the representation. Conjunctions of formulae are represented as ordered lists of indices, and conjunction is implemented as an ordered merge.

Using indices also makes it easy to cache the results of theorem prover calls. This is especially important because when another module causes branching, the predicate abstraction modules often make identical theorem prover calls in each of the resulting branches.

Early detection of inconsistency When computing *succ* for a combination of a number of modules, the naïve algorithm, after collecting shared formulae, computes

successors in each module, and *then* forms the Cartesian product of the results. However, as soon as any one of the modules returns an empty set of successors, we know that the overall Cartesian product will be empty too, and there is no need to invoke *succ* for the remaining modules. With this optimisation the operator $\diamond$ is no longer commutative *in terms of efficiency*, and certain orderings of the modules may be better than others.

Shared formulae are not passed back to their originators In the definition of $succ_\diamond$ (in Definition 3.6.1) each module's successor function is passed the shared information $share_\diamond((a_1, a_2), s, \pi)$ — but this includes, as a conjunct, the information provided by that module itself. This is useless, because each module "already knows" the information it provided, and cannot benefit from being told it again, so we filter out such formulae. However, disabling this optimisation functions as a useful and simple sanity check: if a module is found to be making use of its own shared information, then there is a bug in that module.

Caching of path information In the RETURN-STEP part of our model extraction algorithm (Algorithm Fragment 3.5), we need to repeatedly look for paths from the called procedure's entry point to its return point. Searching the model each time causes a tremendous slowdown, so we have devised a scheme to cache information about these paths, invalidating the cached information when necessary.

4.9 Visualisation features and web interface

For ease of use, abstract models built by the user are accessed through a web browser interface. The models appear as icons in a hierarchy of folders (Figure 4.8). By clicking on a model, the user can either draw that model (or part thereof), or model check it.

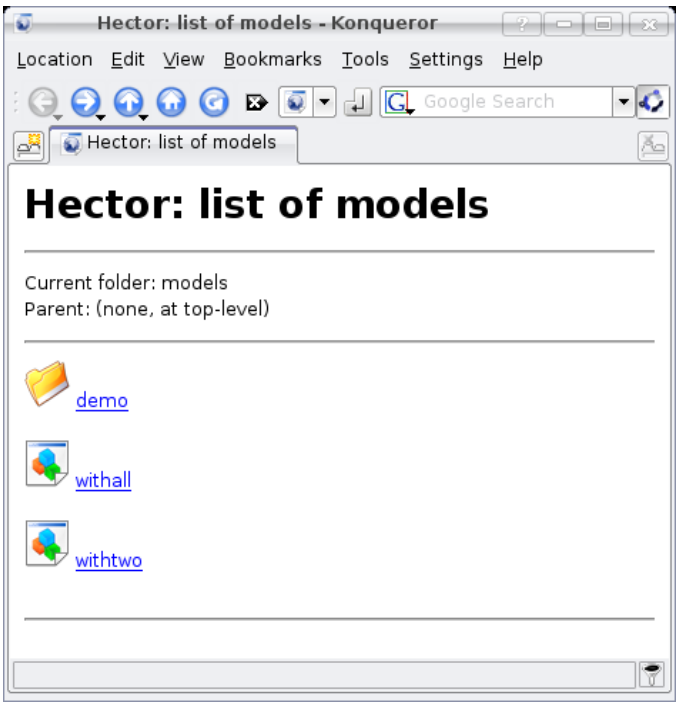

Figure 4.8: Web browser interface through which models are accessed.

4.9.1 Drawing features

Figure 4.9 shows the part of the web interface where drawing options are selected. HECTOR can produce six kinds of drawings of models, of which we will demonstrate three.

- **Control flow graphs only**. Here, as with all the types of drawing, the user can specify which procedures should be included. Figure 4.10 is HECTOR's drawing of our by now familiar square root program.
- **Outline view**. This draws the abstract nodes alongside the CFGs, and the transitions between them, but doesn't show the abstract values associated with each abstract node. See Figure 4.11.
- **Full abstract states**. See Figure 4.12. Here each abstract node is drawn "boxed in" with a depiction of the abstract values for each of the analysis modules in use (here, only **compsigns**). The user can choose to hide the components from particular analysis modules.

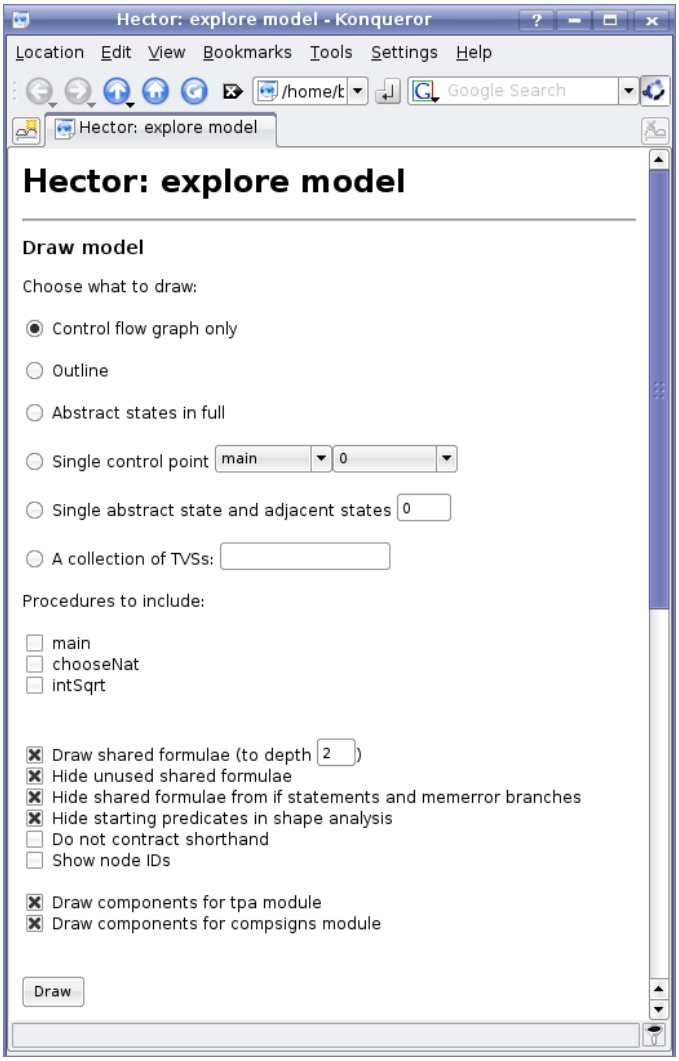

Figure 4.9: Part of the web interface where options for drawing models are selected.

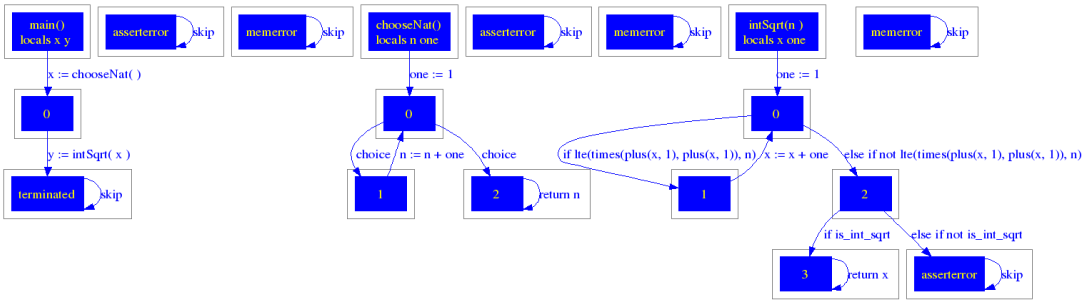

Figure 4.10: Control flow graphs of our square root program, as automatically drawn by HECTOR.

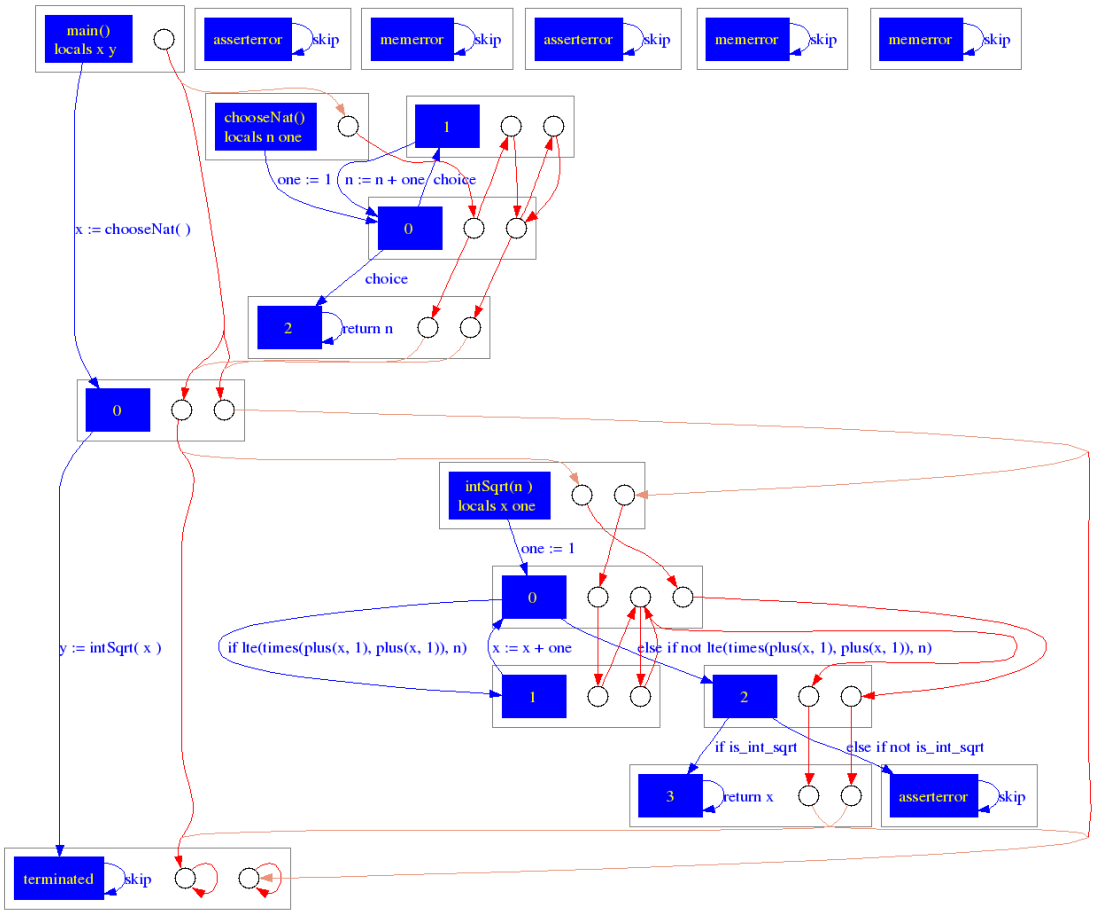

Figure 4.11: Outline view of a model, as automatically drawn by HECTOR. Interprocedural edges are distinguished by their salmon colour.

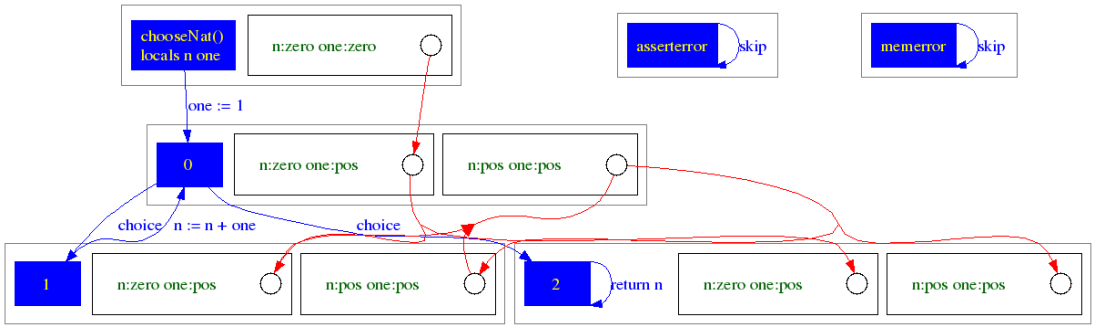

Figure 4.12: Full view of the procedure *chooseNat*, as automatically drawn by HECTOR.

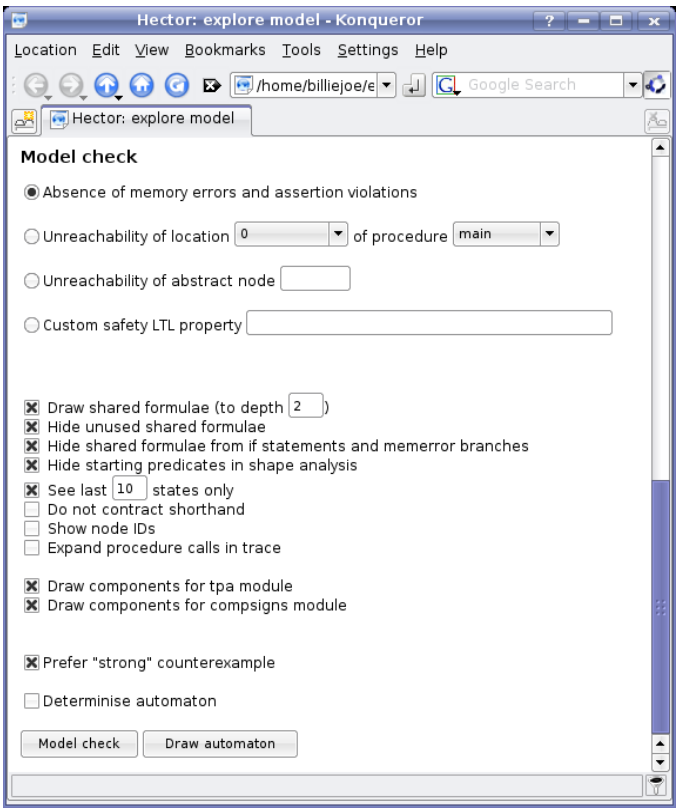

Figure 4.13: Part of the web interface where model checking options are selected.

HECTOR invokes the dot software [GN00] to draw these graphs. The drawing of abstract values is modular too: each analysis module provides a predicate `visualise/4` which renders a given abstract value as a dot subgraph. This is useful for the **tvla** module, which needs to draw its own nodes and edges.

4.9.2 Model checking features

Figure 4.13 shows the part of the web interface where model checking options are selected. HECTOR can search for traces of various kinds. In terms of verification, the interesting option is the first one, "Absence of memory errors and assertion violations", which looks for paths to the special *asserterror* and *memerror* nodes. If this check finds no paths, then the program is verified.

Other options are used to explore the model. Using the "unreachability of location" option we can, for example, search for a trace to the return point (node 3) of the

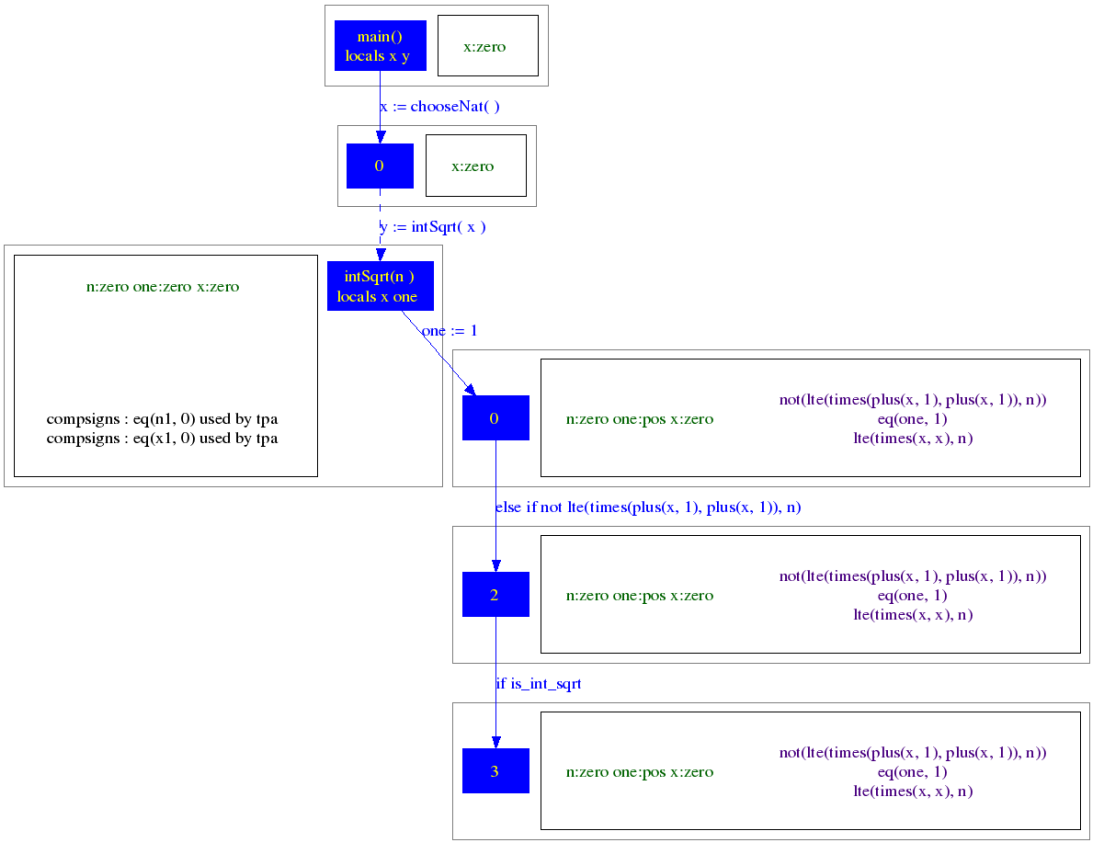

Figure 4.14: HECTOR's drawing of an abstract counterexample trace. (Such traces may in general be infeasible, that is, not correspond to any execution of the concrete program.)

intSqrt procedure. The result is shown in Figure 4.14. Note how the abstract nodes in *intSqrt* are labelled with components for both **compsigns** (sign analysis) and **tpa** (trivector predicate abstraction). Each analysis module uses a different colour to display its abstract values.

Note that traces which are returned in this way are traces through the abstract model, and aren't guaranteed to correspond to an execution in the concrete program; we discussed this on page 44. Currently the user needs to work out whether counterexamples are real ("feasible") or not.

There are two further things of note in Figure 4.14. Firstly, at the entry point of *intSqrt*, HECTOR shows us that sharing proved useful: the useful shared formulae are shown, together with the module that provided them (in this case **compsigns**) and the module that used them (in this case **tpa**). Such sharing information can be hidden if desired. Secondly, the trace "skips over" the call to *chooseNat*, taking the summary edge instead (because this is the shortest route). The option "expand procedure calls in trace" allows us to look inside such calls when required.

4.10 Summary

This chapter described our experimental software model checker HECTOR which implements the module-based verification framework developed in Chapter 3.

We began by outlining the overall architecture of our software, which is written in Prolog, and then detailed how we implemented seven verification/abstraction techniques as analysis modules. These seven are a diverse range of points in the "design space" of verification/abstraction techniques: two kinds of predicate abstraction, a three-valued shape analysis, a type system, symbolic constant propagation, sign analysis and heap reference tracking.

We concluded the chapter with an account of the visualisation features of HEC-

TOR, which are accessed through a web interface, giving examples of the kinds of customisable graphical output that the tool can produce.

In the next chapter we will put our HECTOR implementation to good use, conducting a case study to explore the benefits of automatic modular domain combination.

Chapter 5

Case study

5.1 The MineSweeper game

For our case study, following [LR07], we use HECTOR to verify some properties of an implementation of the popular puzzle game MineSweeper (shown in Figure 5.1).

We give a brief description of the game. The game is played on a grid of cells, some of which contain mines. When the game begins, all cells are *unexposed*, with a number of mines randomly distributed across the grid (Figure 5.1 left). By left-clicking an unexposed cell, the player exposes it. If the newly exposed cell is mined the game is over and the player loses; otherwise the game continues. The player wins by exposing all the unmined cells. Unmined exposed cells display the number of mines in adjacent cells (the *adjacency count*, Figure 5.1 right), and using this information the player decides which cells are safe to expose. If the player deduces that an unexposed cell contains a mine, he may *mark* that cell by right-clicking it. Marked cells, which are displayed with a flag, cannot be exposed (left-clicking them does nothing) which prevents accidental left-clicking. A marked cell can be unmarked by again right-clicking it. Finally there is a button (labelled with a face in Figure 5.1) which ends the game and begins a new one; when the game is over,

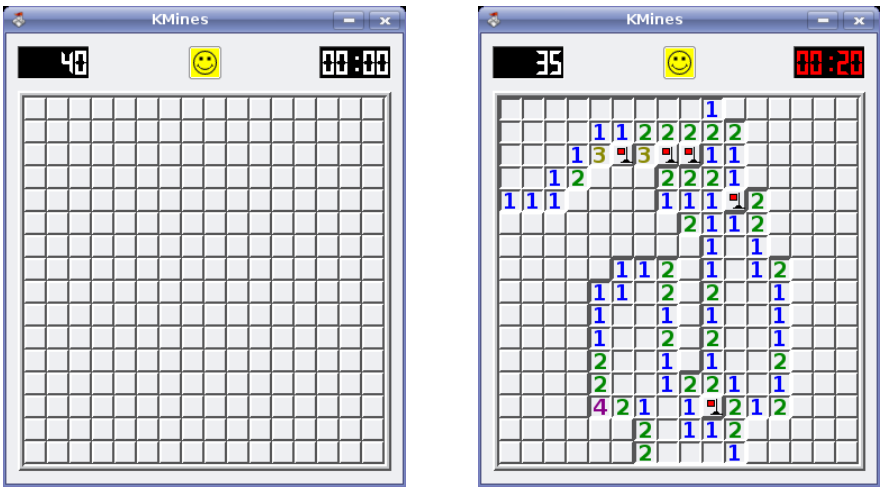

Figure 5.1: The MineSweeper puzzle game, an implementation of which we verify.

this is the only button that has any effect.

A subtlety of the game is that each cell is *automatically exposed* if one of its neighbours becomes exposed and has an adjacency count of zero; clearly an automatically exposed cell cannot contain a mine. Automatic exposure spares the player the tedious task of clicking all the neighbours of cells displaying zero. Note that automatically exposing a cell may trigger further automatic exposures, so that a single left-click might expose in one go a large mine-free region of the grid.

5.2 Our implementation of MineSweeper

Our implementation of MineSweeper works with arbitrarily large game boards, and consists of 356 program statements across 24 procedures. Together with the 48 required declarations (for fields, procedures, and shorthand formulae) and a few comments, this gives an input file of about 800 lines. We chose it for our case study because it uses all the main features on which we would like to test HECTOR: as we will see, the implementation uses linked data structures, pointer arithmetic and recursion. In particular, the program is verifiable neither by first order predicate abstraction alone (because it contains reachability assertions), nor by TVLA alone

(because it uses pointer arithmetic). The program was prepared principally by translating into HECTOR's input language, by hand, the publicly available source code used for the case study from [LR07].

5.2.1 The Model-View pattern

As in [LR07], the idea is that our implementation is split into a Model part and a View part, as per the popular design pattern. The Model part represents the board in memory using a table i.e. a two-dimensional array. Each table entry records whether the cell is mined, whether it is marked, whether it is exposed, and the number of adjacent cells containing mines. The View part uses a linked list structure to keep track of the cells it is displaying to the user; the idea is that whenever the Model part exposes a cell, it should "notify" the View part, by invoking a particular procedure. Thus, an important property for us to verify is *consistency* between the Model and the View, i.e. that the cells in the list of those being displayed to the user are exactly those recorded in the table as being exposed.

(Note that our idealised target language doesn't include any features for modules or information hiding, so that a certain amount of imagination is required to see the boundary between the Model and the View. This is one of the disadvantages of choosing a simplified target language.)

5.2.2 Structure of implementation

Figure 5.2 shows the procedure call graph for our case study program. As usual, self-loops indicate recursive procedures. *Generator* procedures, as described on page 84, are used to non-deterministically generate all possible distributions of mines, for all grid sizes, and all sequences of user actions. This ensures that every possible situation is considered. In Figure 5.2 the generator procedures are indicated by italic names. The role of each procedure is as follows:

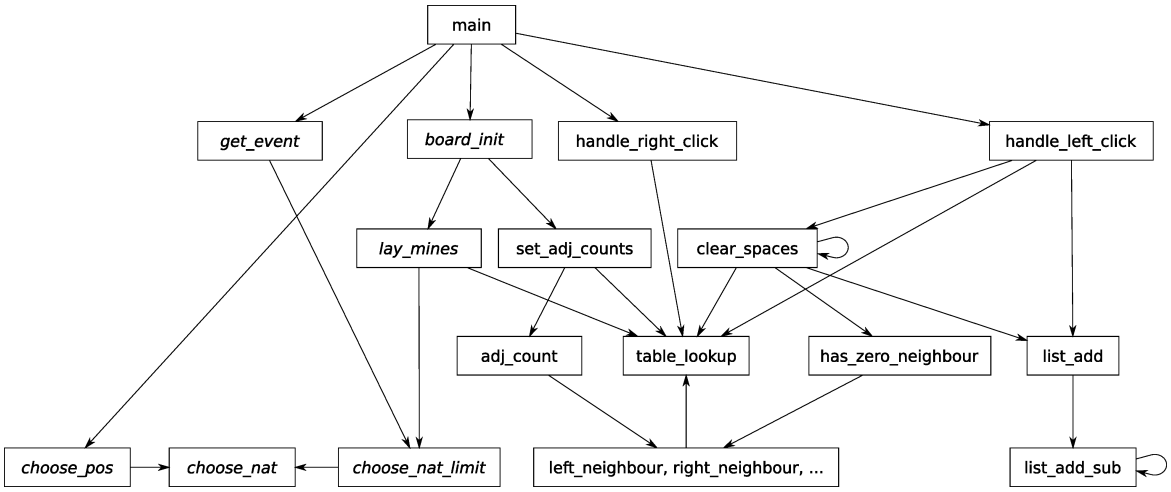

Figure 5.2: Procedure call graph for our case study program. As usual, recursive procedures are shown with a self-loop. Italic names indicate *generator* procedures, which use the `Choice` statement.

main This is the program's main procedure. At the top level, the program sits in an event-handling loop, repeatedly requesting an event object, which encodes the user's next action, and then invoking the appropriate procedure to process the event.

get_event Generates an event object, which records the details of the user's next action: which cell of the grid the user clicked, and with which mouse button.

board_init Sets up the grid for a new game (in all possible ways, nondeterministically), starting with all cells unexposed and unmarked.

board_laymines Distributes a given number of mines across the grid, nondeterministically in all possible ways.

set_adj_counts Once the mines are distributed, this procedure iterates across the grid, correctly setting the adjacency count for each cell.

adj_count Determines and returns the correct adjacency count for a given grid cell.

handle_right_click Handles right-click events, marking and unmarking grid cells as appropriate.

handle_left_click Handles left-click events, exposing unmarked cells when they are left-clicked, and invoking automatic exposure as needed. This procedure records the appropriate cells as exposed in the Model's table representation, and then calls list_add to add them to the View's list of displayed cells. If the user has exposed a mined cell, this is noticed here, and the game is declared over.

clear_spaces Performs automatic exposure of cells as described earlier, again updating both the Model's table representation and the linked list used by the View.

has_zero_neighbour Determines, for a given grid cell, whether any of its neighbours are exposed with a zero adjacency count (and thus whether the cell should be automatically exposed).

list_add Given a linked list and a heap object, appends the object to the end of the list. The object should not be in the list already.

list_add_sub An auxiliary procedure for list insertion.

*Dir*_**neighbour** The procedure left_neighbour returns, for a given cell, that cell's left neighbour in the grid, or null if no such neighbour exists (because the cell lies on the left edge). Similar procedures right_neighbour, top_neighbour etc. exist for the other seven directions.

choose_nat Nondeterministically generates and returns all natural numbers.

choose_pos Nondeterministically generates and returns all positive integers.

choose_nat_limit Nondeterministically generates and returns all natural numbers up to a given limit.

table_lookup Computes, for a two-dimensional array, the memory address of the entry corresponding to given row and column numbers.

5.2.3 Properties we verify

What properties of the implementation should we prove? Obviously we wanted to show that our program doesn't cause memory errors (such as accessing unallocated storage), and HECTOR always looks for such errors. Additionally, however, we wanted to prove some properties which make sense *at the level of the application* and its concepts. We added these as assertions. For instance, in the program's main loop we assert that

$$\mathit{gameover} = 1 \quad \leftrightarrow \quad \exists X.\mathit{cell}(X) \wedge \mathit{isMined}(X) = 1 \wedge \mathit{isExposed}(X) = 1 \qquad (5.1)$$

i.e. that the game is recorded as being over exactly when some mined cell has been exposed. Here we have used HECTOR's shorthand mechanism, where we define $\mathit{cell}(X)$ to be shorthand for

$$X \geq \mathit{board} \quad \wedge \quad X < \mathit{board} + (\mathit{size} \times \mathit{size})$$

which expresses that X is found somewhere in the table representing the game board[1]. (Recall that our shorthand mechanism is purely cosmetic and does not allow one to introduce inductively defined predicates.) We also assert consistency between the Model and the View by

$$\forall X.\big((\mathit{cell}(X) \wedge \mathit{isExposed}(X) = 1) \quad \leftrightarrow \quad \mathit{listElem}(X)\big) \qquad (5.2)$$

where the shorthand $\mathit{listElem}(X)$ uses transitive closure to express membership of X in the linked list maintained by the View:

$$TC_{[A,B]}\big[\mathit{allocd}(A) \wedge \mathit{allocd}(B) \wedge \mathit{next}(A) = B\big](\mathit{explist}, X)$$

[1]The prover Simplify really isn't an oracle, and is defeated if we instead use the following equivalent definition $\exists R \exists C.\big(R \geq 0 \wedge C \geq 0 \wedge R < \mathit{size} \wedge C < \mathit{size} \wedge X = \mathit{board} + (R \times \mathit{size}) + C\big)$.

These high level properties are similar to those verified in [LR07].

But in addition to high-level assertions such as (5.1) and (5.2), we also annotated most of our procedures with assertions at their start and end. For example, for the procedure list_add which appends the parameter *newelem* to the given linked list, at the beginning we assert $allocd(newelem)$ and at the end we assert

$$\forall X.allocd(X) \leftrightarrow allocd_{\mathbf{0}}(X) \quad \wedge \quad \forall X.reachable(X) \leftrightarrow \big(reachable_{\mathbf{0}}(X) \vee X = newelem\big)$$

(where *reachable* and $reachable_{\mathbf{0}}$ are suitable shorthands using transitive closure) which says that during execution of list_add, no new memory was allocated, and the objects in the list at the end are exactly those at the beginning plus the new element.

Annotating each procedure in this way is to a degree redundant, because if some utility procedure doesn't work properly, then this will be reflected by a high-level failure of the overall program. However, we found the extra annotations very useful, because they help to localise the cause of reported errors (be they real or artifacts of abstraction); rather than looking at a counterexample trace showing a high-level failure of the program, and having to work out the cause, we are given a trace ending very soon after something has gone wrong. This helped immensely during the iterative (human-powered) refinement of the configuration of the analysis. Furthermore, such annotations can be reused if the procedure is reused in a new program.

Overall, breaking the assertions into their top-level conjuncts, we ended up with 43 conditions to verify. Recall that these properties are verified for all finite sizes of the game board, however large.

```
Guess an appropriate configuration C
while true do
    Extract model M from program using C
    if program is verified by M then
        return "program is correct"
    else
        Read off counterexample trace T from M
        if T is real error then
            return "program is faulty, as witnessed by T"
        else
            By looking at T determine why configuration C is insufficient
            Improve C to eliminate T
        end if
    end if
end while
```

Figure 5.3: The abstract-check-refine loop. Using this process, one hopes to eventually arrive at a configuration sufficient to verify the program.

5.3 Verification using all analysis modules

We verified our case study program (all 43 conditions) by using six of the seven analysis modules implemented in HECTOR. As mentioned earlier, we developed our model iteratively, essentially following the process shown in Figure 5.3. This activity is called the *abstract-check-refine loop* (e.g. [HJMS02]). (We stress that, at each stage, the formulation of the improved configuration was done by hand; the later Subsection 7.3.2 discusses the prospects of adding automated abstraction refinement to HECTOR.)

In this way, we repeatedly improved the configuration of the analyses until we obtained a configuration which verified all 43 of the asserted properties.

Figure 5.5 summarises this final configuration, as well as the verification run. Figure 5.4 shows an abstract state from this verification, with a component for each of the six analysis modules.

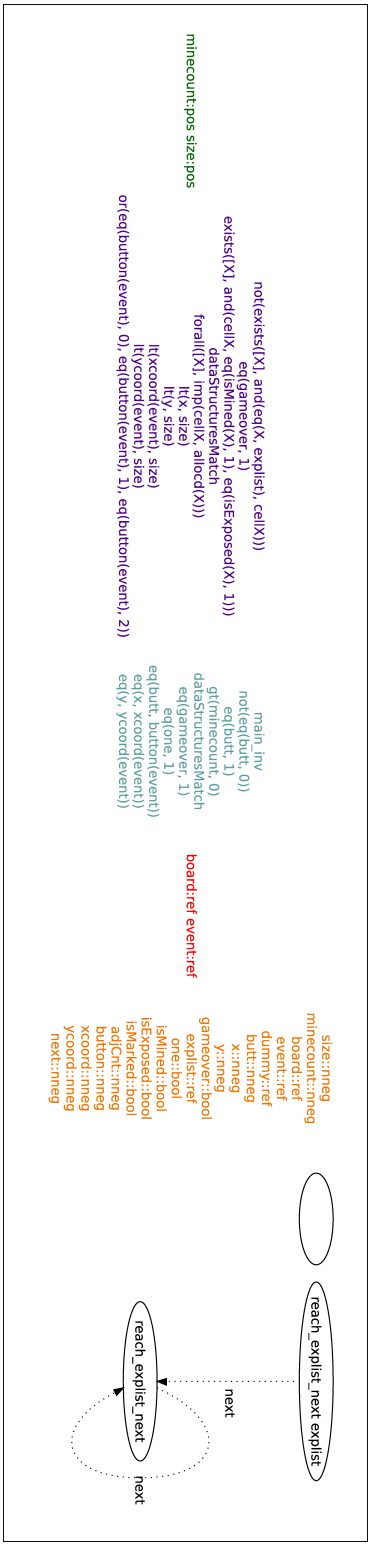

Figure 5.4: An abstract state from our verification, with a component for each of the six analysis modules. From left to right these are: **compsigns**, **tpa**, **symbprop**, **refs**, **types** and **tvla**

.

	No. shared	No. used	%
tvla	115	14	12.17
types	24139	513	2.13
refs	4830	39	0.81
symbprop	12205	258	2.11
tpa	21201	148	0.70
compsigns	2855	27	0.95
Total	67294	999	1.48

Providing module \ Listening module	tvla	types	refs	symbprop	tpa	compsigns
tvla					✓ 14	
types				✓ 16	✓ 504	✓ 1
refs	✓ 24			✓ 10	✓ 5	
symbprop	✓ 30	✓ 16			✓ 222	
tpa	✓ 50			✓ 98		
compsigns				✓ 12	✓ 17	

Summary of configuration

tpa	
Abstraction predicates	123
compsigns	
Variables tracked	30
refs	
Variables tracked	38
tvla	
Variables tracked	11
Fields tracked	1
Instrumentation predicates	5
Sharing patterns	2
Focus declarations	2
Total	212

No. of states in model: 2076
Time to extract model: 374 seconds

Figure 5.5: Summary of the verification using all six analysis modules. The middle table shows, with ticks, between which combinations of modules sharing proved beneficial during EXTRACT-MODEL; the numbers on the ticks indicate how many shared formulae were useful.

	No. shared	No. used	%
tvla	115	14	12.17
tpa	28886	78	0.27
Total	30504	92	0.30

Providing module \ Listening module	tvla	tpa
tvla		✓ 14
tpa	✓ 78	

Summary of configuration

tpa	
Abstraction predicates	292
compsigns	
Variables tracked	0
refs	
Variables tracked	0
tvla	
Variables tracked	11
Fields tracked	1
Instrumentation predicates	5
Sharing patterns	2
Focus declarations	2
Total	313

No. of states in model: 1631
Time to extract model: 437 seconds

Figure 5.6: Summary of the verification using only two of the analysis modules. Because we did not exploit all of our analyses, the time taken for model extraction is longer, and the amount of user configuration required is substantially higher.

5.4 Comparison: verification using two modules only

For purposes of comparison, we also verified the same program (again all 43 conditions) using a model constructed with only two of HECTOR's analysis modules, namely those for shape analysis and trivector predicate abstraction. The configuration used, along with the resulting verification, is summarised in Figure 5.6.

We can see that, as we hoped, the model built using all analysis modules required much less configuration (i.e. much less user guidance), and the model was built in a shorter time. Specifically, the model extraction time was cut by 64 seconds, a reduction of 14%. 101 fewer configuration items were needed, a reduction of 32%.

The increase in speed occurs because the additional analysis modules provide specialised, streamlined mechanisms for reasoning about certain properties e.g. references, signs of integers etc. and thus avoid the relatively expensive theorem prover calls which predicate abstraction uses. This is particularly true of the type system which, instead of doing more work at each successor computation, requires only a single flow-insensitive analysis of the program.

The decrease in user input required is mainly due to the type system and symbolic constant propagation modules. The type system module needs no configuration because it performs type inference, and the symbolic constant propagation module likewise automatically identifies properties that are likely to be of use.

Obviously we have shown such an improvement only for a single target program, but because HECTOR automatically generates summaries of the kind shown in Figures 5.5 and 5.6, we have the necessary machinery in place to support a thorough investigation of this effect.

(When we compare the total configuration sizes, we are implicitly assuming that each kind of configuration item is equally difficult for the user to provide. We do

this for simplicity, but it probably isn't exactly the case; e.g. in general writing an abstraction predicate is more taxing than naming a variable for the sign analysis to track. In any case, our conclusion is not threatened: if the six-module verification requires less configuration when treating each item with equal weight, this will be *even more* the case if we give lower weights to the simpler configurations for **compsigns** and **refs**.)

Comparison: our case study vs. Hob case study

In [LR07], an implementation of MineSweeper is used as a case study for the Hob verification system. We will discuss the Hob system in general in Subsection 7.2.2, and compare it with HECTOR, but for now we make some brief remarks concerning the two systems' respective verifications of MineSweeper.

The MineSweeper implementation we have used was obtained by translating into HECTOR's input language, by hand, the publicly available source code from the Hob case study. The properties we verify are very similar to those verified using Hob. The Hob system [KLZR05, LKR05] also allows different analysis techniques to cooperate in order to verify a program, but the mechanism by which domains interact is quite different. In particular, with Hob each procedure is analysed by a single domain; the domains interact only at procedure boundaries. In such a system the opportunities for the domains to improve each others' results are limited, and one cannot make beneficial use of shallow domains as we do in HECTOR. The Hob verification uses PALE (discussed on page 64), rather than TVLA, to perform the shape analysis.

5.5 Some interesting uses of propagation

In this section we highlight some of the situations, observed in our verifying models, in which the propagation of formulae between the analysis modules plays a crucial role. To simplify the presentation, we will focus on the two-module model, and thus restrict ourselves to interactions between the predicate abstraction and shape analysis modules; as Figure 5.5 shows, in the six-module model there are interactions between many pairs of modules. Also, we will only present a high-level "story" of what the sharing achieves, and will omit much of the detail.

We consider the following scenario of MineSweeper play.

1. A game is in progress. In the event-handling loop in 'main', the procedure 'get_event' is called to return the player's next action.

2. The returned event turns out to be a left-click, so 'handle_left_click' is called, passing as parameters the grid coordinates of the clicked cell.

3. 'handle_left_click' calls 'table_lookup', which performs pointer arithmetic, to get the table entry for the clicked cell.

4. The clicked cell turns out to be unmarked and unexposed, so 'handle_left_click' exposes it (in the Model's table) by setting the 'isExposed' field to 1.

5. The newly exposed cell must now be added to (the View's) linked list of cells to display, which 'handle_left_click' does by calling 'list_add'.

6. 'handle_left_click' now calls 'clear_spaces' to perform any necessary automatic exposure of cells, but none is needed.

7. 'handle_left_click' returns, passing control back to the event-handling loop in 'main', ready for the player's next action.

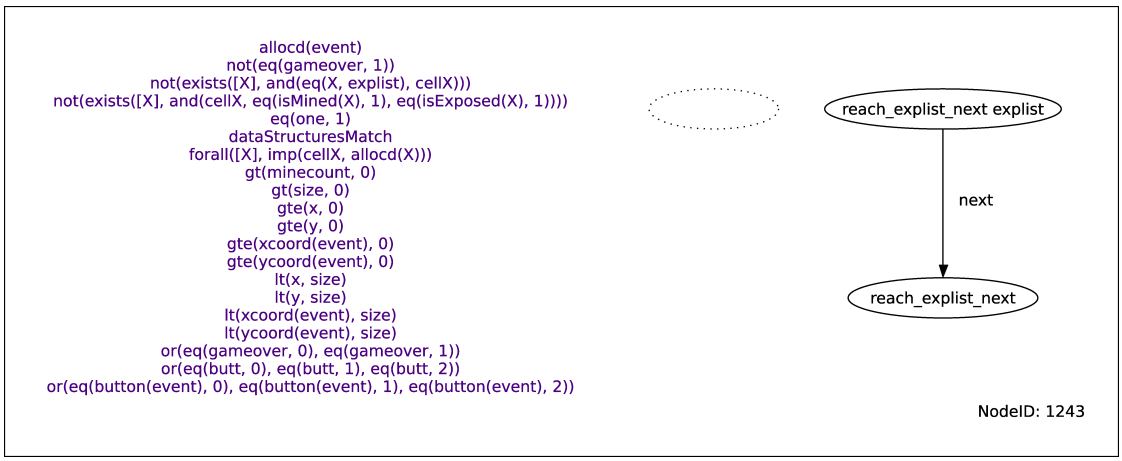

Figure 5.7: The abstract state at the beginning of our scenario.

Figure 5.7 shows one of the abstract states occurring at the start of the event-handling loop, as drawn by HECTOR. In this particular state, we see that the linked list contains exactly one element (not counting the dummy head node); the summary node represents all of the rest of the heap. Here *dataStructuresMatch* is shorthand for the consistency property (5.2) (page 192).

Figure 5.8 shows the abstract state reached in 'handle_left_click', just before the invocation of 'table_lookup'. Notice that the predicate abstraction component contains, among others, the following facts:

1. $x < size$
2. $y < size$
3. $x \geq 0$
4. $y \geq 0$
5. $\forall X.cell(X) \rightarrow allocd(X)$

The first four of these say that x and y are in the right range to be grid coordinates, and the fifth records that the memory range corresponding to the Model's table is all allocated. It is these facts which allow the predicate abstraction to determine

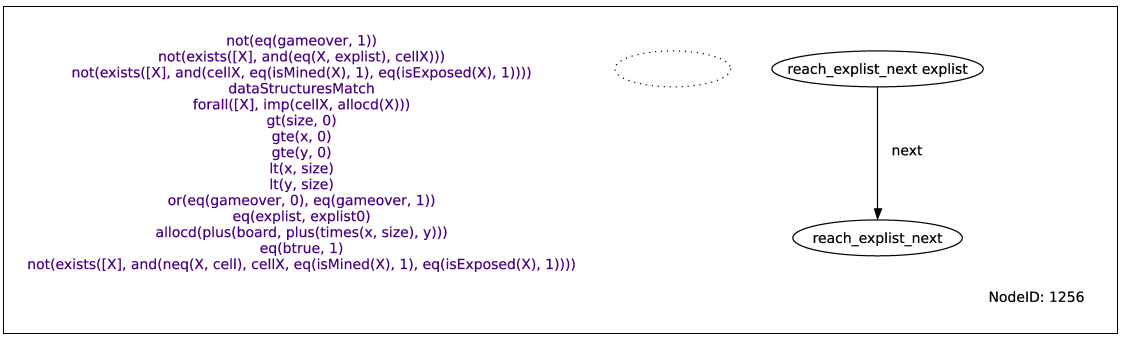

Figure 5.8: An abstract state encountered during our scenario, just before 'table_lookup' is invoked to compute (using pointer arithmetic) the address of the table entry for the clicked grid cell.

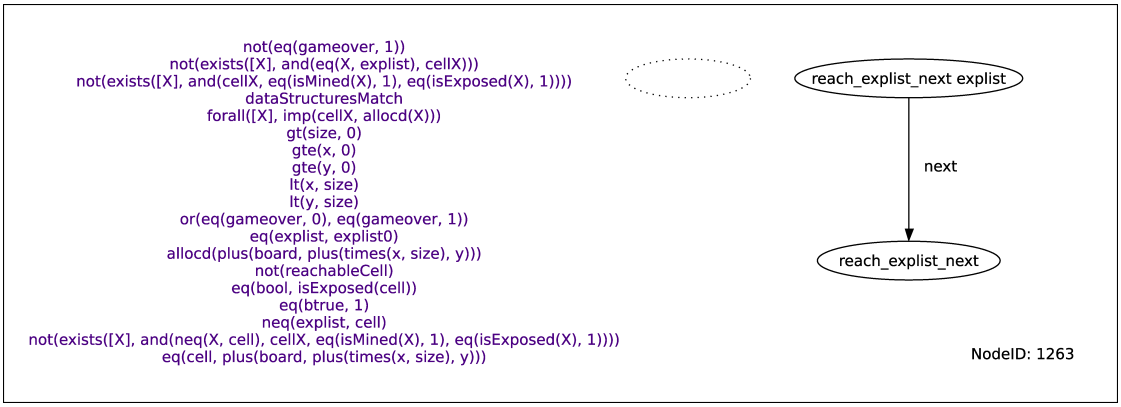

Figure 5.9: An abstract state encountered during our scenario, after 'handle_left_click' has examined the clicked cell and found it to be unmarked and unexposed; importantly, the predicate abstraction module has now established the fact $\neg$*reachable(cell)*.

that, after the pointer arithmetic performed by 'table_lookup', the address in the variable *cell* is allocated.

In our scenario, 'handle_left_click' then examines the clicked cell and finds it to be unmarked and unexposed; the abstract state after this is shown in Figure 5.9. Importantly, the predicate abstraction module has now established the fact $\neg$*reachable(cell)*, which is shorthand for

$$\neg\, TC_{[A,B]}\big[allocd(A) \wedge allocd(B) \wedge next(A) = B\big](explist, cell)$$

This is derived as follows (informally): from *dataStructuresMatch* we see that *cell*

Figure 5.10: An abstract state encountered during our scenario, after the clicked cell's *isExposed* field has been set to 1, but before the cell has been added to the linked list.

has its *isExposed* field set to 1 iff it is present in the linked list — but we have just examined the isExposed field, and found it not to be 1. Note that although this produces a fact involving TC, it doesn't rely on any genuine TC reasoning; our predicate abstraction module's treatment of TC subformulae as uninterpreted predicates suffices here.

Next, 'handle_left_click' exposes the clicked cell. Figure 5.10 shows the state after the cell's *isExposed* field has been set to 1 (for the Model) but before the cell has been added to the linked list (for the View). The property *dataStructuresMatch* has been broken, but instead the predicate abstraction module proves the fact

$$\forall X.(cell(X) \wedge isExposed(X) = 1) \leftrightarrow (listElem(X) \vee X = cell)$$

The abstract state on entry to 'list_add' is shown in Figure 5.11 (top half); the object to be added, *cell*, is passed in via the *newelemx* parameter.

The first action of 'list_add' is to copy the address from the variable *newelemx*, which is ignored by the shape analysis and had its value computed by pointer arithmetic, to the variable *newlem* which the shape analysis tracks. As explained in Section 4.5.2, a pure shape analysis such as we use must in general "give up" when faced with this situation, because arithmetic can give rise to values which are neither null nor

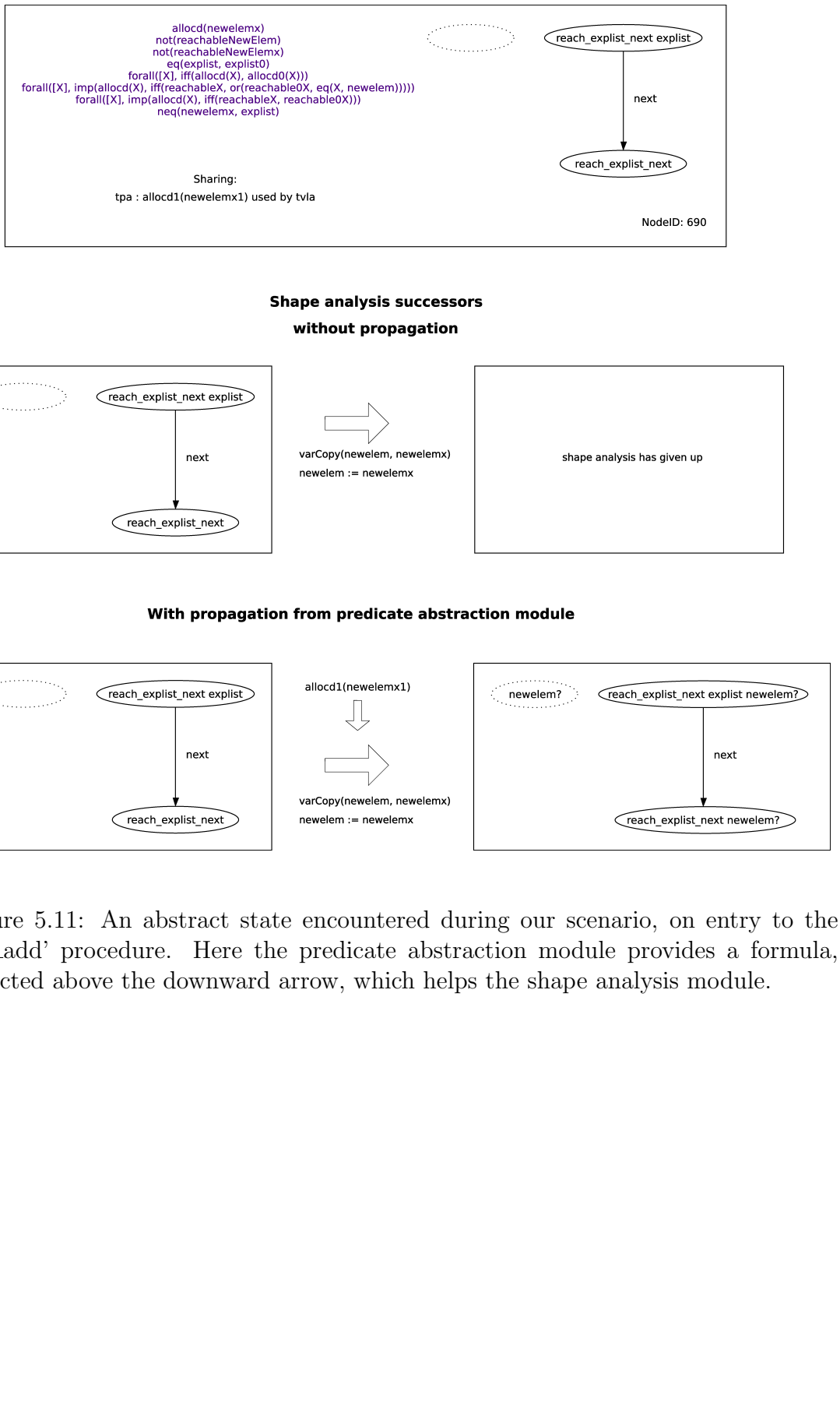

Figure 5.11: An abstract state encountered during our scenario, on entry to the 'list_add' procedure. Here the predicate abstraction module provides a formula, depicted above the downward arrow, which helps the shape analysis module.

a valid pointer (e.g. negative integers, unallocated addresses).

This is the first place where propagation between the analysis modules helps. Our predicate abstraction module can provide the fact

$$allocd_{\mathbf{I}}(newelem_{\mathbf{I}})$$

which enables the shape analysis to continue, treating *newelem* as a pointer that can potentially point to any heap object. This is depicted in Figure 5.11 (bottom half).

In the next successor step, the shape analysis module focuses the variable *newelem* to find out exactly where it points. Without sharing, five possible heaps are generated, three of which would prevent a successful verification. However, information propagated from the predicate abstraction module eliminates these problematic heaps. The first is eliminated because it has *newelem* as null which contradicts *allocd*(*newelem*), the second because it has *newelem* equal to the dummy list head node which contradicts $newelem \neq explist$, and the third because it has *newelem* already present in the list, which contradicts $\neg reachable(newelem)$. This is shown in Figure 5.12.

The auxiliary procedure 'list_add_sub' walks along the linked list until it finds the last existing element, and then tacks on the new element with the statement `curr.next := newelem`. Figure 5.13 shows the abstract states before and after the execution of the statement `curr.next := newelem`. We see that the shape analysis module has propagated the formula

$$\forall X.allocd(X) \rightarrow \big(reachable(X) \leftrightarrow (reachble_{\mathbf{I}}(X) \vee X = newelem)\big)$$

which expresses the fact that all objects that were in the list before the statement are still in it, and additionally *newelem* is now an element. The predicate abstraction module (which, recall, performs no reachability reasoning of its own) uses this fact

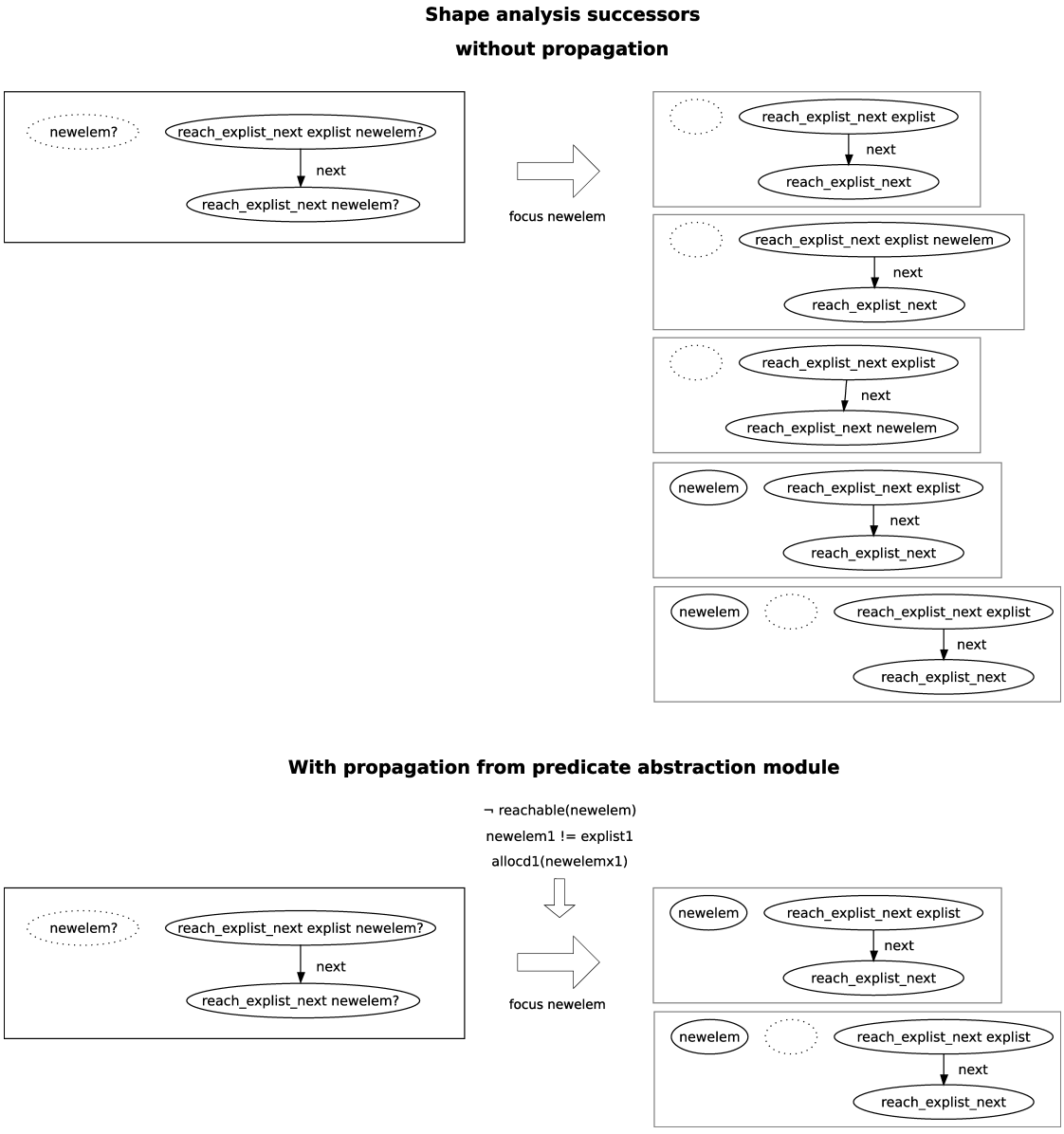

Figure 5.12: Part of an abstract state encountered during our scenario, in the 'list_add' procedure. Here the predicate abstraction module provides three formulae, listed above the downward arrow, which help the shape analysis module, allowing it to return fewer successors.

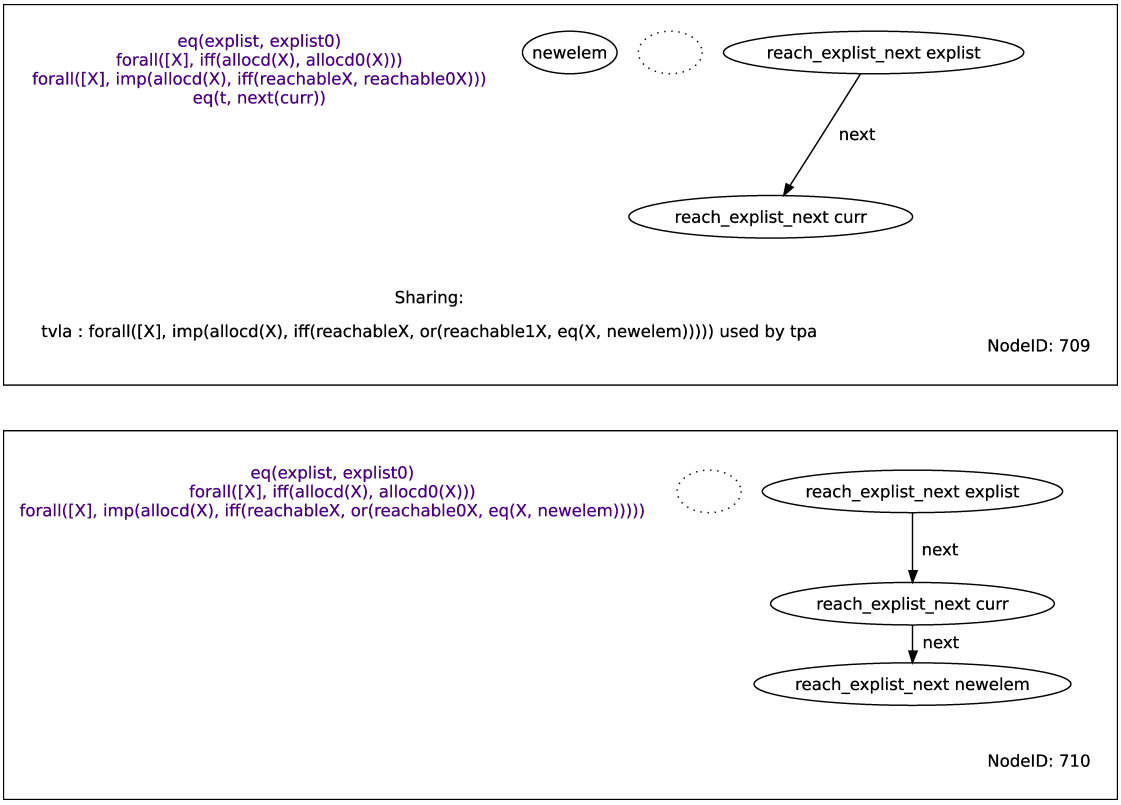

Figure 5.13: Abstract states encountered during our scenario, respectively before and after the statement `curr.next := newelem` which adds the new element to the end of the linked list. Here the shape analysis module is "returning the favour", providing a fact about reachability that helps the predicate abstraction module.

to establish

$$\forall X.allocd(X) \rightarrow \big(reachable(X) \leftrightarrow (reachble_{\emptyset}(X) \vee X = newelem)\big)$$

This second formula, which looks similar, expresses the effect on the list of *all statements since entry to the current procedure*, and, when the call to 'list_add' returns, will be essential in proving that the property *dataStructuresMatch* has been restored.

5.6 Real counterexample for a false property

Originally we also included the following assertion in our program's main loop:

$$\neg\exists X.cell(X) \wedge isExposed(X) = 1 \wedge isMarked(X) = 1$$

This says that there is never a cell which is both marked and exposed. This would be true, we reasoned, because while a cell is marked left-clicking it has no effect and thus one cannot expose the cell; on the other hand once a cell becomes exposed, it cannot be subsequently marked because right-clicking it has no effect.

When we added this assertion to our configuration as an abstraction predicate, HECTOR gave us a counterexample, which we quickly concluded was a real error — not in the program, but in the asserted property. The trace showed the property holding up until a call to the clear_spaces procedure, and failing to hold afterwards. The clear_spaces procedure performs automatic exposure as described earlier, which we had overlooked, and this can indeed expose a marked cell, if one of its neighbours becomes exposed and has a zero adjacency count.

5.7 Summary of case study

In this chapter, we reported on our verification of an implementation of the well-known MineSweeper puzzle game, which uses linked data structures, pointer arithmetic and recursion. This program can be verified neither by conventional predicate abstraction (which cannot handle linked structures) nor by TVLA shape analysis (which doesn't handle pointer arithmetic). We showed, however, that using HECTOR we could verify the program, because our predicate abstraction and shape analysis modules work cooperatively. This is a good success for our approach and demonstrates Benefit B1 from our Introduction.

Another success for HECTOR is that when we also used our four lightweight modules alongside those for predicate abstraction and shape analysis, the time taken to verify the program substantially decreased, as did the amount of user guidance needed. This demonstrates benefit B2.

The significance of our case study is that it shows that our approach can work in practice, and that the benefits B1 and B2 of our approach are not merely things which look good on paper — they really do happen.

Chapter 6

Additional model checking features

In Chapter 4 we described features of HECTOR which produce graphical output — for instance these can display the model, or a specified part thereof, and draw counterexample traces, and the output can be customised in various ways. These features work well when dealing with toy example programs. While conducting the case study of Chapter 5, however, we found that when the model has several thousand states, even with these features it becomes difficult to find one's way around the model.

In this chapter we describe two extensions to the basic HECTOR system which aim to make dealing with large models easier:

1. **Model checking for a safety fragment of LTL:** We allow the user to find traces they are interested in by specifying them in a temporal logic, namely a "two-level" safety fragment of LTL.

2. **Post-pruning of the model:** Sometimes an abstract state is generated which contains a "buried" inconsistency, which only "rises to the surface" after sev-

eral further execution steps. We give an algorithm which traces such inconsistencies back as early as possible, pruning away extraneous states.

We also describe a third extension:

3. **Falsifying programs:** The machinery developed in Chapter 3 allows us to verify programs by showing that the error locations cannot be reached. Here we develop a method, working on exactly the same abstract models, which sometimes allows the falsification of programs too, by showing that an error location is definitely reached.

Extension 3 doesn't relate directly to the case study, and we planned it independently; however, we include it here because it turns out to use exactly the same underlying mechanism as Extension 2.

6.1 LTL model checking

A (linear) temporal logic is a formal language for specifying properties of execution traces of a system. Typically the user writes a formula specifying *the negation of* the kind of execution traces he is interested in, and enters this formula into a model checking program. The model checker then either produces a trace of interest from the system, or states that no suitable trace exists.

6.1.1 Our temporal logic

The temporal logic we use, which is a slight adaptation of *syntactically safe LTL* [Sis94], is given by the following grammar:

$$\Phi \quad ::= \quad \Phi \wedge \Phi \mid \Phi \vee \Phi \mid X\Phi \mid G\Phi \mid \Phi W \Phi \mid \text{atomic} \mid \neg \text{ atomic}$$

$$\text{atomic} \quad ::= \mathit{atLoc}(\pi, l) \mid \mathit{atNode}(n) \mid \mathit{inProc}(\pi) \mid p$$

where $p \in \mathscr{L}$. Such formulae are evaluated along execution paths. The connectives $\wedge, \vee, \neg$ have their usual meanings. The temporal formulae $X\Phi$, $G\Phi$ and $\Phi_1 W \Phi_2$ mean, respectively: Φ is true at the next state of execution, Φ is true at all future states, and Φ_1 is true until Φ_2 becomes true (or Φ_1 is true forever — this is a weak until). The above grammar builds only safety properties, that is, properties whose falsity can be demonstrated by a finite execution trace.

What about the atomic formulae? It is suggested in [GJ02, YRS01] that, to apply temporal logic to abstraction-based software verification, we should let the propositions of the language *themselves be constructed according to an appropriate first-order grammar*; thus we allow any formula p from $\mathscr{L}$ to be used as a proposition. We also add some propositions for "navigation" around the model: $\mathit{atLoc}(\pi, l)$ is true just when execution is at the location l of procedure π, $\mathit{atNode}(n)$ is true just when execution is at the abstract node with ID number n, and $\mathit{inProc}(\pi)$ is true just when execution is in the procedure π. (Note however that the above safety language does *not* allow temporal connectives within the scope of a first-order quantifier, so we cannot track a particular object from one state to the next. While this can be done [YRSW03, Dis03] and is clearly sometimes a useful thing, it does make the situation very much more complicated.)

Suppose we want to look at an execution in which, during the procedure 'handle_left_click', the value of *gameover* changes from 0 to 1. Such an execution is characterised by the LTL formula

$$F(\mathit{inProc}(\mathit{handle\_left\_click}) \wedge \mathit{gameover} = 0 \wedge X \neg \mathit{gameover} = 0)$$

which we can write in our fragment as the negation of

$$G(\neg \mathit{inProc}(\mathit{handle\_left\_click}) \vee \neg \mathit{gameover} = 0 \vee X\, \mathit{gameover} = 0)$$

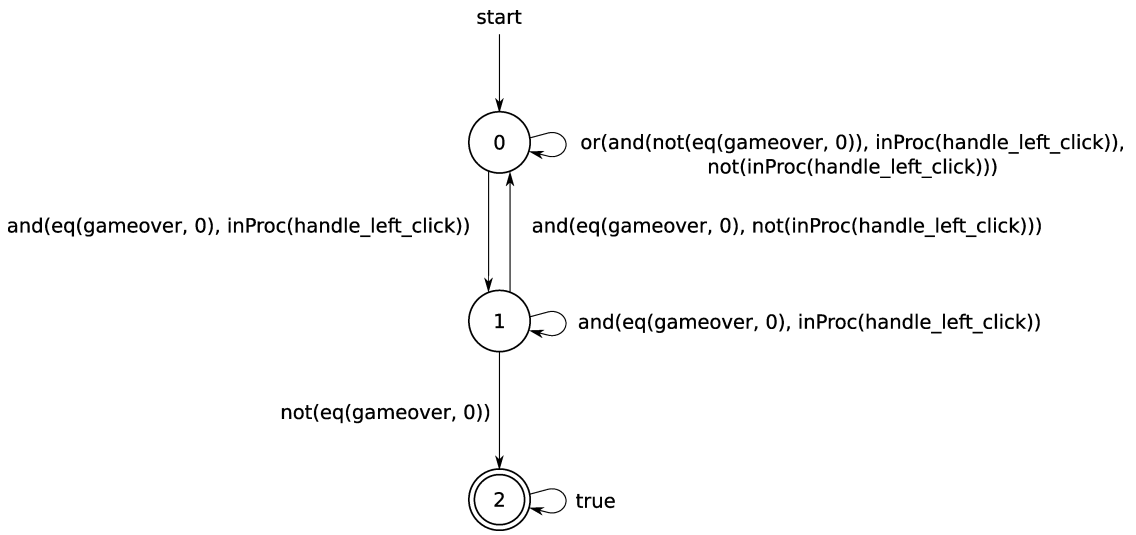

Figure 6.1: An automaton generated from a temporal logic formula.

By invoking the existing 'scheck' software described in [Lat03], HECTOR translates temporal formula to automata on finite words. (In general LTL requires the more complicated Büchi automata, but we are working only with safety properties so we can, and prefer to, avoid them.) Figure 6.1 shows the automaton generated for the above formula. The 'scheck' program never sees the first-order (i.e. $\mathscr{L}$) subformulae — HECTOR replaces them with fresh proposition names before running 'scheck' and then performs the inverse substitution on the resulting automaton.

We then perform model checking in a largely standard way — as described in [KV01], conceptually, we incrementally compute the product of our model and the formula's automaton, and look for paths leading to the automaton's accepting state. Figure 6.2 shows a suffix (the last four nodes) of the trace which HECTOR returns for the query above (showing only the **tvla** components). We emphasise that the "proposition" $gameover = 0$ is not an abstraction predicate in the model; this is no problem for the model checker.

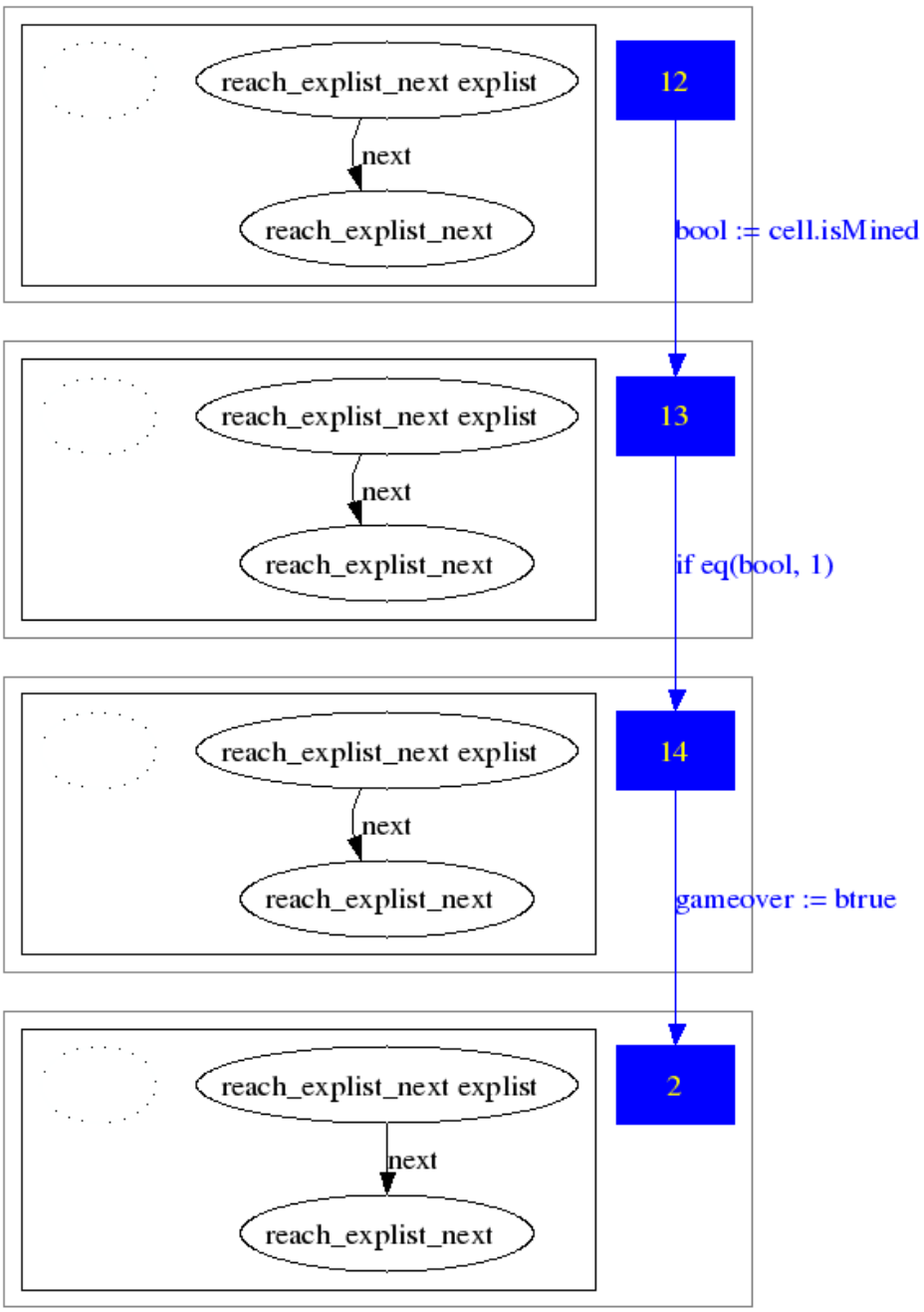

Figure 6.2: An execution trace found by HECTOR in response to our temporal logic query. (Here we have asked for the last four states only, and the components for the **tvla** module only.)

6.1.2 Evaluating arbitrary $\mathscr{L}$-formulae in arbitrary abstract states

We have so far omitted one important detail, however. When computing the product of the automaton and model, one must be able to check, for some state n in the model, whether n satisfies some formula Φ labelling an edge in the automaton — this is how one decides which transitions in the product are possible and which are not. In systems without abstraction, this can be done precisely. For instance, if the model is an explicitly represented Kripke structure and the labels are propositional formulae, one simply looks up the required propositions in the required model state and evaluates. But in our case, the model's states are abstract, and the edge labels can include arbitrary formulae from the logic $\mathscr{L}$, which is not even decidable.

Our approach is to reuse the formula sharing machinery we have already built, to conservatively but imprecisely evaluate $\mathscr{L}$-formulae in abstract states, as follows:

procedure EVAL($a : T$, $\pi : \mathit{ProcNames}$, $\Phi : \mathscr{L}$)
    $\mathit{Succs} := \mathit{succ}(a, \texttt{Skip}, \pi, \Phi)$
    if $\mathit{Succs} = \emptyset$ **then**
        return definitely-false
    else
        return maybe-true
    end if
end procedure

Informally, we model a `Skip` statement, putting the formula Φ in as an extra constraint. If no successors are returned, this indicates that an inconsistency was detected between Φ and the abstract state a, and thus Φ must be false in all concrete states represented by a.

On the other hand, if some successors are returned, this does not guarantee the existence of a concrete state represented by a and satisfying Φ, due to the over-

approximating nature of the *succ* operation.

Running time of model checking The above evaluation procedure EVAL, and hence the whole model checker, can be quite slow; for instance, evaluating our example query in the two-module model takes 126 seconds. This is still very favourable, however, compared to the time it might take the user to find a suitable trace by manual inspection of the model. We expect that formulae with more complicated (e.g. quantified) "propositions", such as

$$G\big(\neg \mathit{inProc}(\mathit{list\_add}) \quad \vee \quad \neg(\forall X.\, \mathit{allocd}(X) \rightarrow \mathit{reachable}(X))\big)$$

will generally take longer to check.

6.1.3 Is our temporal logic checking procedure sound?

We believe that the model checking procedure outlined above is sound, in the sense that if the model checking procedure cannot find a trace which satisfies a given temporal formula Φ, then no such trace exists in the (concrete) program. We believe this because it seems correct theoretically, and because our implementation seems to behave as we would expect. We stress however that we haven't yet done any of the formal development needed to state and prove such a soundness result. (This result would resemble Criterion 3.3.4, which proved that our method of showing particular program locations unreachable is sound.)

Also left as future work is formulating a "thorough" semantics for LTL over our abstract models; this problem is trickier than it first appears, and it is not clear what form such a semantics should take.

6.1.4 Sources of loss of precision

Although we believe it to be sound, our model checking procedure is imprecise — it can return counterexamples which either do not correspond to any execution in the concrete program, or which correspond only to concrete executions which don't in fact falsify the temporal formula in question. There are four sources of this imprecision:

1. As we saw in the previous subsection, our evaluation of the automaton's guard conditions is imprecise, returning either 'definitely-false' or 'maybe-true'.

2. The transition relation we calculate between abstract states is over-approximate, due to the over-approximate nature of *succ*.

3. Our abstract states may have empty concretisations, that is, may not correspond to any concrete state.

4. For simplicity we treat the model as a plain transition system, rather than a pushdown system, which means that we sometimes produce counterexample traces in which the procedure calls and returns do not match up.

6.2 Falsifying safety properties

In this section we present a new method which allows the user to *falsify* safety properties as well as verify them.

First, let us recall the basic idea which underlies the verification methods of HECTOR and systems such as SLAM [BR01] and BLAST [HJMS02]. Given a program, these systems build a model which over-approximates the program: intuitively an over-approximating model has all the behaviours of the original program, and possibly many more. Hence, any "bad" behaviours present in the program are also present

in the model, and therefore if the model contains no bad behaviours, the program does not either.

However, this scheme does not allow the *falsification* of safety properties, because bad behaviours found in the model need not be present in the original program — they may be "artifacts" introduced by the over-approximation process, and therefore "not feasible" in the original program. Existing approaches to the falsification of safety properties have focused on showing that abstract counterexamples are indeed feasible, for example by:

1. searching for a corresponding concrete counterexample (e.g. [PDV01]),
2. proving the feasibility of the abstract counterexample path by satisfiability checking (e.g. [BR01]), or
3. adding under-approximation or calculation of "must-transitions" to the model (e.g. [GHJ01, GC06]).

The new falsification method we present here uses only over-approximating models. In particular, our method doesn't perform any of the tasks (1), (2), (3) listed above. Instead, our method is inspired by *generalised model checking* (GMC) [BG00] and exploits the seriality of programs, i.e. the fact that the execution of a program does not just "stop" for no reason.

Remark 6.2.1. Seriality of concrete semantics. Let $\pi, s_0 \xrightarrow{p} l, s$ and let $\mathit{Edges}(\mathit{Graph}(\pi))(l)$ take any of the forms except `Call` and `Return`. Then there exist l', s' such that $\pi, s_0 \xrightarrow{l,s} l', s'$. □

Later, we will study the cost of our new falsification check and find that, although it is potentially very imprecise, the check essentially "comes for free", so we believe it is a worthwhile feature.

6.2.1 H: a judgement for falsification

For falsification, our queries will be of the same kind as we used in Subsection 3.3.3 for verification: they will be given by a set $B \subseteq \mathit{ProcNames} \times \mathit{Locs}$ of "bad" locations, and we wish to find out whether any of these locations are reached by the program, i.e. whether

$$\exists (\pi, l) \in B \;\; \exists s_0, s, p \quad \pi, s_0 \xrightarrow{p} l, s$$

We introduce a new judgement $\mathbf{H}\pi, l, a$, which we can think of as meaning "The abstract node (π, l, a) is Hopeless", i.e. once execution reaches (π, l, a) there is no hope for avoiding forever the set of bad locations B. The named derivation rules for this judgement are as follows (leaving the dependence of $\mathbf{H}$ on B implicit):

h-already-there

$$\frac{(\pi, l, a) \in N \qquad (\pi, l) \in B}{\mathbf{H}\pi, l, a}$$

h-choice-1

$$\frac{\mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Choice} : l'_1 : l'_2 \qquad \forall a' \in \mathscr{A},\ \text{if } (\pi, l, a)\mathscr{R}(\pi, l'_1, a') \text{ then } \mathbf{H}\pi, l'_1, a'}{\mathbf{H}\pi, l, a}$$

h-choice-2

$$\frac{\mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Choice} : l'_1 : l'_2 \qquad \forall a' \in \mathscr{A},\ \text{if } (\pi, l, a)\mathscr{R}(\pi, l'_2, a') \text{ then } \mathbf{H}\pi, l'_2, a'}{\mathbf{H}\pi, l, a}$$

h-call

$$\frac{\mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Call}(u, \pi', [p_1, \ldots, p_k]) : l' \qquad \forall a' \in \mathscr{A},\ \text{if } (\pi, l, a)\mathscr{R}^{\mathit{call}}(\pi', \mathit{start}, a') \text{ then } \mathbf{H}\pi', \mathit{start}, a'}{\mathbf{H}\pi, l, a}$$

h-return

$$\frac{\begin{array}{c}\mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Return}(v) \\ \forall a' \in \mathscr{A}, \forall l' \in \mathit{Locs},\ \text{if } (\pi, l, a)\mathscr{R}^{return}(\pi', l', a') \text{ then } \mathbf{H}\pi', l', a'\end{array}}{\mathbf{H}\pi, l, a}$$

h-ord

$$\frac{\begin{array}{c}\mathit{Edges}(\mathit{Graph}(\pi))(l) \text{ has any other form} \\ \forall a' \in \mathscr{A}, \forall l' \in \mathit{Locs},\ \text{if } (\pi, l, a)\mathscr{R}(\pi, l', a') \text{ then } \mathbf{H}\pi, l', a'\end{array}}{\mathbf{H}\pi, l, a}$$

The h-already-there rule says that, if execution is already at a bad location, then trivially execution cannot avoid the bad locations forever. The rules h-call, h-return and h-ord (for calls, returns and ordinary statements respectively) are based on seriality: our abstraction does not tell us to which of the successors execution will flow, *but it must flow somewhere*, and if *all* the possibilities are hopeless, then the current node is hopeless too. Crucially `Choice` statements are treated differently: because both branches of the nondeterministic choice are possible, it is enough that *in one branch* all the successors are hopeless (the other branch is allowed to have non-hopeless successors).

The following theorem shows that what we are calling "hopeless" nodes really do inevitably lead to bad locations.

Theorem 6.2.2. Given a program P and a sound abstract model of P, and a model checking problem B, if

$$\text{(A.)}\quad \mathbf{H}\pi, l, a \qquad \text{(B.)}\quad \pi, s_0 \xrightarrow{p} l, s \qquad \text{(C.)}\quad (s_0, s) \in \gamma(a)$$

then

$$\text{(D.)}\qquad \exists(\pi', l') \in B\ \ \exists s'_0, s', p'\quad \pi', s'_0 \xrightarrow{p'} l', s'$$

Proof: We proceed by structural induction on the derivation of $\mathbf{H}\pi, l, a$. So let (A.),

(B.), (C.) be true. There are six cases to check, corresponding to the six derivation rules.

h-already-there In this case we have $(\pi, l, a) \in N$ and $(\pi, l) \in B$. From (B.) we have $\pi, s_0 \xrightarrow{p} l, s$. To obtain (D.) as required, simply set: $\pi' = \pi$, $l' = l$, $s'_0 = s_0$, $s' = s$ and $p' = p$.

h-choice-1 The premises of h-choice-1 give us

$$\text{(X.)} \qquad \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Choice} : l'_1 : l'_2$$

and

$$\text{(Y.)} \qquad \forall a' \in \mathscr{A},\ \text{if } (\pi, l, a)\mathscr{R}(\pi, l'_1, a') \text{ then } \mathbf{H}\pi, l'_1, a'$$

Now (B.) and (X.) meet the premises of the choice-1 rule, so we derive the conclusion $\pi, s_0 \xrightarrow{l,s} l'_1, s$. By sound-intra, we see that there exists $a' \in \mathscr{A}$ such that: $(\pi, l'_1, a') \in N$, $(s_0, s) \in \gamma(a')$ and $(\pi, l, a)\mathscr{R}(\pi, l'_1, a')$. Instantiating (Y.) gives us $\mathbf{H}\pi, l'_1, a'$, which we can use as (A.) in the induction hypothesis giving the conclusion (D.) as required.

h-choice-2 Almost identical to the previous case.

h-call The premises of h-call give us

$$\text{(X.)} \qquad \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Call}(u, \pi', [p_1, \ldots, p_k]) : l'$$

and

$$\text{(Y.)} \qquad \forall a' \in \mathscr{A},\ \text{if } (\pi, l, a)\mathscr{R}^{call}(\pi', \mathit{start}, a') \text{ then } \mathbf{H}\pi', \mathit{start}, a'$$

Using (B.) and (X.) we invoke the call-1 rule to obtain $\pi, s_0 \rightarrow l, s : \pi', s'$. By sound-call, there exists $a' \in \mathscr{A}$ such that $(\pi', \mathit{start}, a') \in N$, $(s', s') \in \gamma(a')$ and also $(\pi, l, a)\mathscr{R}^{call}(\pi', \mathit{start}, a')$.

Using the call-2 rule, we have $\pi', s' \xrightarrow{\epsilon} \mathit{start}, s'$, and instantiating (Y.) gives us $\mathbf{H}\pi', \mathit{start}, a'$; using these as (B.) and (A.) respectively in the induction hypothesis gives the conclusion (D.) as required.

h-return This is the trickiest case. The premises of the h-return rule give

$$\text{(X.)} \qquad \mathit{Edges}(\mathit{Graph}(\pi))(l) = \texttt{Return}(v)$$

and

$$\text{(Y.)} \qquad \forall a' \in \mathscr{A}, \forall l' \in \mathit{Locs},\ \text{if } (\pi, l, a)\mathscr{R}^{\mathit{return}}(\pi', l', a') \text{ then } \mathbf{H}\pi', l', a'$$

We cannot be in the main procedure (i.e. $\pi \neq \pi_1$) because the main procedure doesn't include a return, so by inspection of the rules for semantics of programs, we see that the derivation of (B.) $\pi, s_0 \xrightarrow{p} l, s$ must at some point have used the call-2 rule to obtain $(\pi, s_0 \xrightarrow{\epsilon} \mathit{start}, s_0)$, and therefore we must have had the premise $\pi', t_0 \rightarrow l_1, t_1 : \pi, s_0$. This in turn can only have been obtained with the call-1 rule, so we also had (among others) the premise

$$\mathit{Edges}(\mathit{Graph}(\pi'))(l_1) = \texttt{Call}(u, \pi, [p_1, \ldots, p_k]) : l_c$$

We can now invoke the return rule, and we find that there exists t_c such that

$$\pi', t_0 \xrightarrow{l_1, t_1, \pi, s_0, l, s} l_c, t_c$$

Now by sound-return there exists $a' \in \mathscr{A}$ such that $(\pi', l_c, a') \in N$, $(t_0, t_c) \in \gamma(a')$ and also $(\pi, l, a)\mathscr{R}^{\mathit{return}}(\pi', l_c, a')$. Instantiating (Y.) we get $\mathbf{H}\pi', l_c, a'$ which we can use as (A.) in the induction hypothesis giving the conclusion (D.) as required.

h-ord The ord rule covers all the other statement forms. The premises of ord

give us

$$(\text{Y.}) \qquad \forall a' \in \mathscr{A}, \forall l' \in \mathit{Locs},\ \text{if } (\pi, l, a)\mathscr{R}(\pi, l', a') \text{ then } \mathbf{H}\pi, l', a'$$

By Remark 6.2.1 (seriality) we find that there exist l', s' such that $\pi, s_0 \xrightarrow{l,s} l', s'$. From sound-intra there exists $a' \in \mathscr{A}$ such that: $(\pi, l', a') \in N$, $(s_0, s') \in \gamma(a')$ and $(\pi, l, a)\mathscr{R}(\pi, l', a')$. Instantiating (Y.) gives us $\mathbf{H}\pi, l', a'$, which we can use as (A.) in the induction hypothesis giving the conclusion (D.) as required. □

We can now justify falsification using judgement **H**.

Criterion 6.2.3. Given a program P and a sound abstract model of P, and a set of bad nodes B, then

$$(1.) \qquad \forall a \in \mathscr{A},\ \text{if } (\pi_1, \mathit{start}, a) \in N \text{ then } \mathbf{H}\pi_1, \mathit{start}, a$$

is sufficient to verify

$$(2.) \qquad \exists (\pi, l) \in B\ \ \exists s_0, s, p \quad \pi, s_0 \xrightarrow{p} l, s$$

i.e. is sufficient to show some bad node is reached.

Proof: We will use Theorem 6.2.2, putting $\pi = \pi_1$, $l = \mathit{start}$ and $s = s_0 = s^{\mathit{start}}$, to obtain the required conclusion. By sound-init, there exists $a \in T$ such that $(\pi_1, \mathit{start}, a) \in N$ and $(s^{\mathit{start}}, s^{\mathit{start}}) \in \gamma(a)$. This immediately gives us (C.) $(s_0, s) \in \gamma(a)$. Applying the init rule obtains $\pi_1, s^{\mathit{start}} \xrightarrow{\epsilon} \mathit{start}, s^{\mathit{start}}$ which is (B.), and instantiating (1.) above gives $\mathbf{H}\pi_1, \mathit{start}, a$ which is (A.). □

6.2.2 Example of falsification

Figure 6.3 shows a variant of our square root program, this time all in one procedure for simplicity. The single procedure nondeterministically generates a natural number

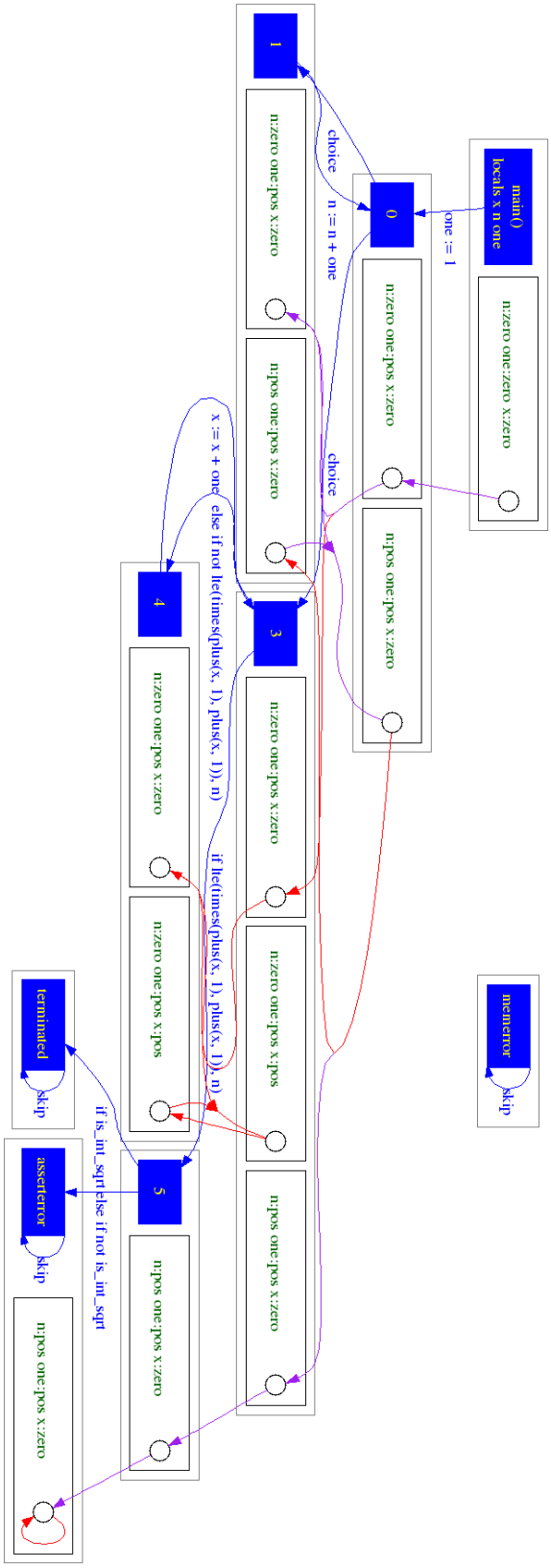

Figure 6.3: An example falsification of a faulty program (our square root program, made faulty by negating the loop guard) using our new falsification method. Edges used in the computation of **H** are shown in purple.

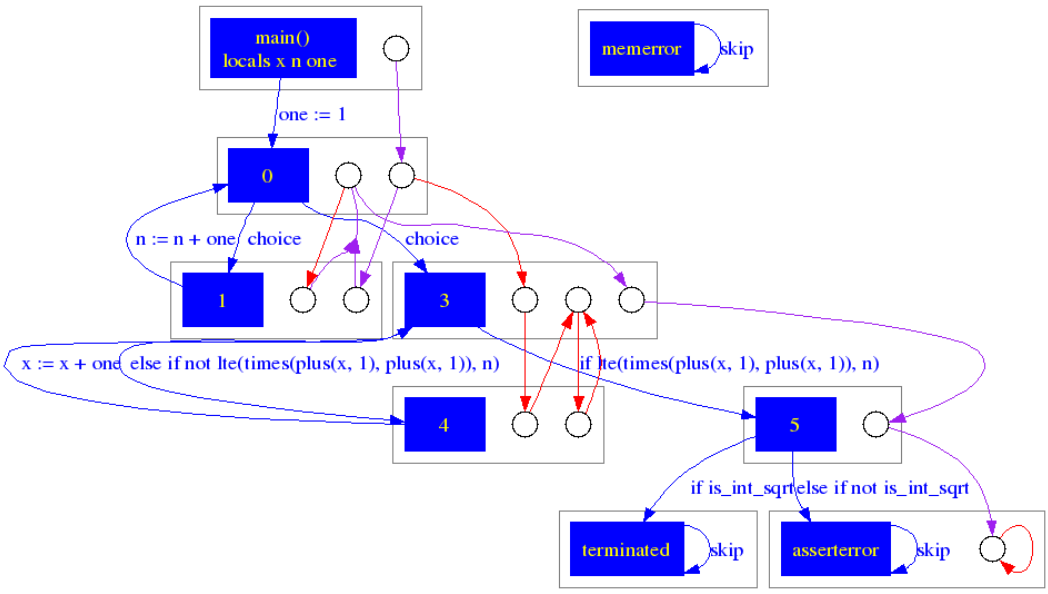

Figure 6.4: Outline view of a falsification of a faulty program. Edges used in the computation of **H** are shown in purple.

n and then calculates its integer square root in x. This time, however, we have introduced a "mistake": the guard for the loop is the negation of what it should be. Our new falsification method can prove that this program reaches the *asserterror* state using sign analysis: in Figure 6.3 the edges used in the computation of **H** are shown in purple. On the outline view (Figure 6.4) this is easier to see. (Note that this falsification would not be possible if `Choice` statements were treated in the same way as other intraprocedural statements.)

6.2.3 Remarks

As stated earlier, this approach to falsification was inspired by GMC [BG00] which, relative to a fixed model, obtains more precise model checking results. GMC achieves this by "case splitting" on unknown propositions; here we similarly case split on each node's set of possible outgoing transitions.

Our paper [CH08] presents this falsification method in a slightly more general way, and uses a different expository device, for which we lack the space here: the approach

is presented as solving a two-player attractor game between two players F, who is trying to Falsify the program, and P who is trying to Prevent this from happening. A position in this game is simply an abstract node (π, l, a) and a move consists of choosing an abstract successor of the current node, which then becomes the new game position. Player F wins by forcing play to a location in B, which falsifies the program. [CH08] presents further methods and analyses which apply when the programming language contains no nondeterministic statements other than `Choice`; this is not the case in the setup of this thesis, because our memory allocation statement is also nondeterministic. In particular, an approach is given which combines our game-based method with that based on under-approximating must transitions. On the other hand, [CH08] omits the treatment of procedures and does not make the connection to post-pruning, which is the subject of the rest of this chapter.

Cost of new falsification check We can compute the full set of abstract nodes n for which $\mathbf{H}n$ (which corresponds to finding the winning region in the associated game) in time $O(m+e)$, where m is the number of nodes in the model, and e is the number of edges. In fact, this set can be computed in a way that visits each edge at most once. This makes our falsification check very fast. Also, note that no changes to the model extraction phase were needed. Therefore, although the check may be very imprecise, it is essentially "free", and therefore worth having.

6.3 Post-pruning models

6.3.1 Paths which "fizzle out"

Figure 6.5 shows HECTOR's "outline" view of the *list_add* procedure in our case study. What is interesting about this graph is that three of the executions, shown in purple, *simply stop* — the final node has no successors. At first this seems to contradict the seriality we relied on in the previous section for falsification.

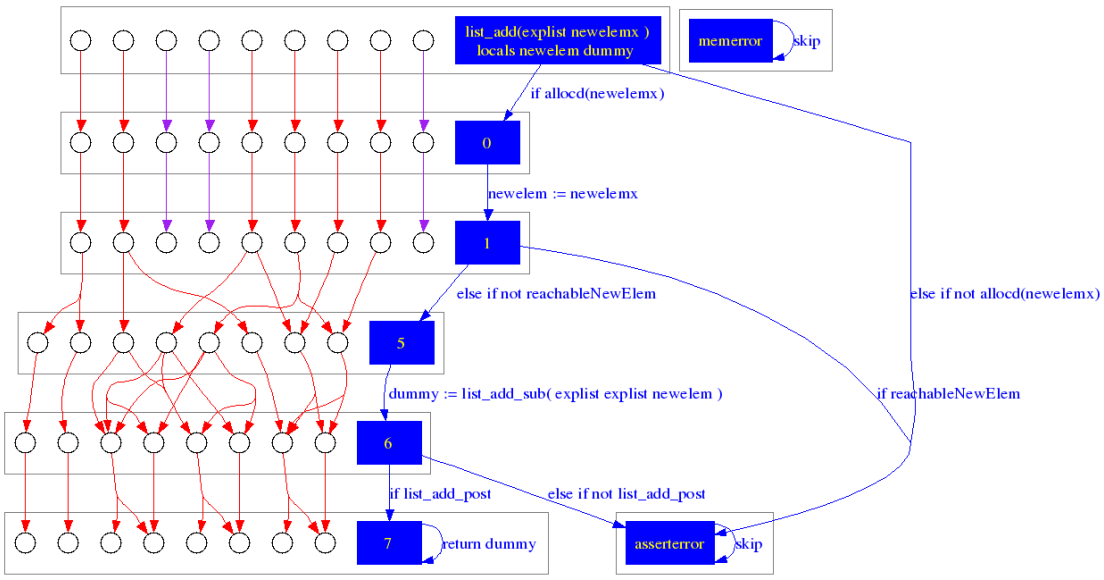

Figure 6.5: Outline view of the procedure *list_add* showing (in purple) three paths which fizzle out.

When one looks more closely at one of these executions, however, as in Figure 6.6, it soon becomes apparent what is going on. The first abstract state of the Figure is already inconsistent: the predicate abstraction and shape analysis parts cannot be reconciled. The predicate abstraction part says that *newelemx* is allocated but is not reachable (i.e. is not in the linked list). But the shape analysis part insists that *every* heap node is reachable (i.e. is in the list). However, this inconsistency is not "noticed" immediately, in this case because the shape analysis part doesn't track the variable *newelemx*. Two statements later, however, when the value is copied from *newelemx* to *newelem*, the inconsistency is detected and no successors are returned.

Subtrees of the model such as this are said to *fizzle out*, and are automatically infeasible. Their existence is irritating for two reasons.

- The extra, infeasible states clutter up the model and make it more difficult for the user to follow.
- The extra states may be explored by our LTL safety checking procedure (which is thus slowed down) and may also be returned as part of counterexamples,

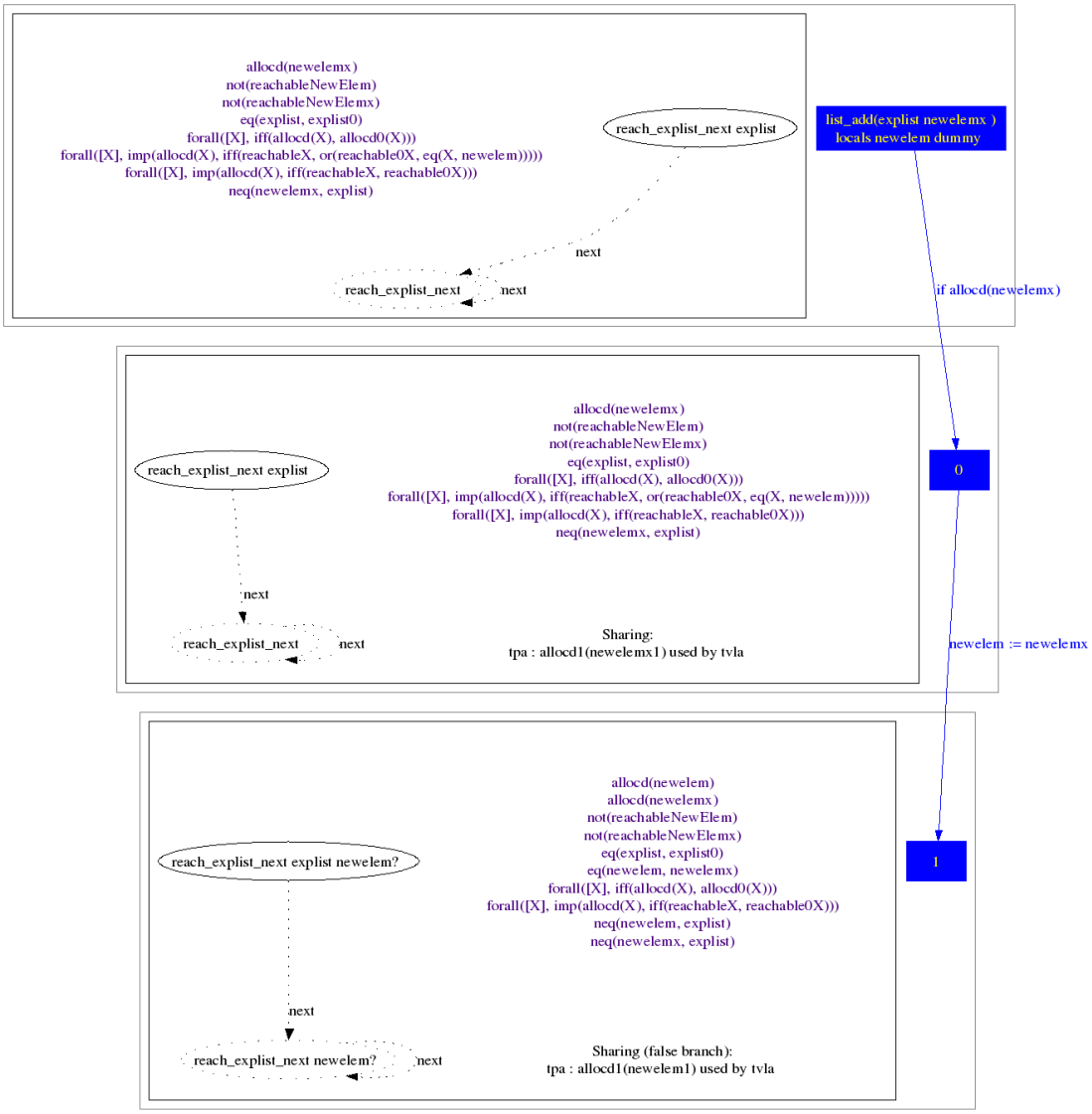

Figure 6.6: An execution path which fizzles out. In the first (topmost) state, the components for **tvla** and **tpa** are already inconsistent, but this is not noticed until the value from *newelemx* is copied into *newelem*.

which makes model checking less precise (because such a counterexample is automatically infeasible).

Here we give a simple method for removing such infeasible parts of the model, thereby making the model smaller and easier to understand, and making our temporal logic model checking procedure more precise and faster.

6.3.2 The post-pruning algorithm

It turns out that we can detect such extraneous states by invoking the falsification algorithm of Section 6.2 with an empty set $B := \emptyset$ of bad locations. To see this intuitively, suppose that for $B = \emptyset$ we have $\mathbf{H}n$ for a node n. This tells us that if execution ever reaches n, it must inevitably at some stage reach the empty set of locations. Of course this is impossible, so we conclude that execution can never reach n. The following lemma captures this intuition.

Lemma 6.3.1. Given a program P and a sound abstract model of P, if $\mathbf{H}n$ for some node $n = (\pi, l, a)$, with respect to the empty set of bad nodes $B = \emptyset$, then there do not exist s_0, s, p such that $(s_0, s) \in \gamma(a)$ and $\pi, s_0 \xrightarrow{p} l, s$.

Proof: Suppose for a contradiction that s_0, s, p as above exist. We will invoke Theorem 6.2.2: by assumption we have $(s_0, s) \in \gamma(a)$, $\pi, s_0 \xrightarrow{p} l, s$ and $\mathbf{H}(\pi, l, a)$ which are (C.), (B.) and (A.) respectively. The conclusion of Theorem 6.2.2 tells us that there exist $(\pi', l') \in B = \emptyset$ which gives us our contradiction. □

We can now state our post-pruning operation.

Definition 6.3.2. Post-pruning operation. Let

$$Q = (T, \gamma, N, \mathit{Edges}, \mathit{CallEdges}, \mathit{ReturnEdges})$$

be an abstract model for a program P. Then defining $X \subseteq N$ by

$$X \quad := \quad \{n \in N \mid \text{ not } \mathbf{H}n \text{ with respect to } B = \emptyset\}$$

we define the pruned model $\mathit{prune}(Q)$ to be

$$(T, \gamma, N \cap X, \mathit{Edges} \cap (X \times X), \mathit{CallEdges} \cap (X \times X), \mathit{ReturnEdges} \cap (X \times X))$$

□

The following theorem, which completes this chapter, shows that our pruning operation is sound.

Theorem 6.3.3. Let Q be an abstract model of a program P (as per Definition 3.3.1). If Q is sound (as per Definition 3.3.2) then $\mathit{prune}(Q)$ is also sound.

Proof: We must show that the properties sound-init, sound-intra, sound-call and sound-return hold for $\mathit{prune}(Q)$. We will only do the case for sound-intra, as the other cases are similar.

Let $\pi, s_0 \xrightarrow{l,s} l', s'$ with $(\pi, l, a) \in N \cap X$ and $(s_0, s) \in \gamma(a)$. By sound-intra for Q, there exists $a' \in T$ such that $(\pi, l', a') \in N$, $(s_0, s') \in \gamma(a')$ and $((\pi, l, a), (\pi, l', a')) \in \mathit{Edges}$. All we need to do, then, is prove that (π, l', a') is in X. So we suppose not and derive a contradiction.

It follows from $(\pi, l', a') \notin X$ that $\mathbf{H}(\pi, l', a')$ with respect to $B = \emptyset$. Then by Lemma 6.3.1, there do not exist $\hat{s_0}$, $\hat{s}$, $\hat{p}$ such that $(\hat{s_0}, \hat{s}) \in \gamma(a')$ and $\pi, \hat{s_0} \xrightarrow{\hat{p}} l, \hat{s}$. But we already have such values, namely s_0, s' and (l, s) respectively. □

6.4 Summary

In this chapter we described three extensions to the basic HECTOR system, which increase its usefulness (particularly when dealing with large models):

1. **Checking of temporal safety properties:** Users can find traces of interest to them (in abstract models that are already built) by making ad-hoc queries in a sophisticated temporal logic.

2. **Post-pruning of the model:** Parts of the model that are inconsistent can sometimes be identified and pruned away "for free", making the model smaller and easier to understand, and producing more precise verification results.

3. **Falsifying programs:** Programs can be falsified (proved incorrect) as well as verified, by using a falsification check. Two-player games provide the intuition for this check, which does not require any changes to the model construction part of HECTOR.

This chapter brings us to the end of the technical contributions of this thesis. In the next and final chapter, we will reflect on what this work has achieved, and the extent to which the goals we set out in Chapter 1 have been met. By reflecting in this way, we will identify important questions to be born in mind if one wishes to continue this line of research.

Chapter 7

Conclusions

7.1 Contributions of this thesis

In Chapter 1 we set a series of objectives for this thesis, in three groups: Design objectives (D1-D4), Implementation objects (I1-I5) and Experimental objectives (E1-E5). We also outlined, at a higher level, the Benefits (B1-B4) that we hoped our approach would provide. We shall now revisit these objectives and proposed benefits (reprinting them in a slightly abbreviated form) which will allow us to review what we have accomplished and what remains as future work.

7.1.1 Revisiting our design objectives

D1 *Fix the "target programming language", i.e. the language in which the programs to be verified will be written, and formalise its syntax and semantics.*

In Section 3.1 we gave a syntax and semantics for our target language. The language is simple and idealised, and untyped, but expressive with recursive procedures and dynamically allocated heap objects. Syntax-wise, we chose to work directly with control flow graphs to avoid the nuisances of source code, and we chose a small but sufficient set of statement forms. We gave operational

semantics for our language, handling recursion directly in the rules rather than with explicit stacks.

D2 *Define the single common language which analysis modules will use to exchange information.*

In Section 3.2 we gave syntax and semantics for a rich logic over program states, which we called $\mathscr{L}$, which analysis modules use to share information about the program state. In fact $\mathscr{L}$ performs triple duty, also being used to express guards for alternation and iteration statements in programs, and assertions about desired program behaviour.

$\mathscr{L}$ incorporates *time indices*, so that e.g. $x_{\mathbf{0}}$ denotes the value of variable x on entry to the current procedure, and $x_{\mathbf{1}}$ denotes the value of x before the current statement executed. Thus $\mathscr{L}$ can express the effects of individual statements, and frame conditions for procedures.

In an important design decision we chose $\mathscr{L}$ to be a first order logic with transitive closure. We gave arguments for this choice in Section 3.7: we noted that some mechanism for expressing reachability in the heap is clearly necessary, and that transitive closure seems to be adequate for many purposes (as opposed to, say, second order logic) while admitting a reasonably clean sound (but incomplete) axiomatisation in first order logic.

In retrospect, in this thesis we have not tested our choice of $\mathscr{L}$ very well, because all our shape reasoning is done by TVLA, and the transitive closure operator is "native" to TVLA. A better test would be to implement new analysis modules based on graph grammars and separation logic, which don't have a native treatment of transitive closure, and see how well $\mathscr{L}$ works as a communication language for such modules.

D3 *Devise an appropriate interface through which analysis modules can talk to the central "broker".*

In Subsection 3.4.1 we gave an interface which our analysis modules present to the central broker. An analysis module $\mathbf{M}$ appears like a "conventional" abstraction domain, with two differences. Firstly, in order to provide information to other modules, there is an extra function $\mathbf{M}.share$ which returns $\mathscr{L}$-formulae which are entailed by a particular abstract value. Secondly, in order to make use of formulae provided by other modules, the abstract successor function $\mathbf{M}.succ$ now takes an $\mathscr{L}$-formula as an extra argument.

In Definition 3.4.3 we gave conditions specifying what it means for an analysis module to be sound; these conditions are relied upon in later proofs.

Overall we found this interface to work well when designing modules, with two minor complaints:

- Occasionally we found that sharing of formulae during procedure calls and returns, which we did not implement, would have worked well. Sometimes, for instance, we had to insert an extra `Skip` statement between successive procedure calls to make time for a round of formula sharing. The limited TC axioms we added to Simplify (page 162) were only needed because of the absence of sharing during calls and returns, and there were also occasions when extra abstraction predicates had to be added to the configuration for the same reason.

  This was not at all surprising to us, as we had believed all along that sharing during calls and returns would be useful; indeed we used it in our earlier publications [Cha06c, Cha06a]. We later dropped this feature because it increases the amount of code needed to implement each analysis module, and complicates various proofs, but is not different in any interesting way from sharing during ordinary statements; it is simply "more of the same".

- The *share* function is passed the appropriate abstract value a, the current procedure name and the statement being executed, but *not* the current

program location. This seemed reasonable because the shared formulae are generally extracted from a. However, it precludes an implementation of the non-null types from [JMG+02]. The idea of non-null types is to identify variables which, despite starting off as null (because all variables do, on procedure entry), are quickly assigned a non-null value and thereafter remain non-null. Such types (and thus the resulting *share* function) need to take program locations into account.

What we have not done, and perhaps should have done, is to ask a third party programmer, who is not familiar with our broker code, to implement an analysis module based solely on seeing the interface. We believe the feedback provided by such a programmer would help us understand whether we have the "right" interface.

D4 *Formulate a generic verification algorithm for the broker, which works with whatever set of analysis modules is presented, and propagates information between them so that they cooperate.*

In Section 3.5 we gave an algorithm EXTRACT-MODEL for automatically extracting models from programs. Our algorithm is worklist-based and uses procedure summarisation to handle (recursive) procedure calls without requiring that procedures be annotated with pre- and post-conditions. Our algorithm is module-based or "generic" in that it works with any analysis module provided, accessing such modules only through the defined interface.

We proved that EXTRACT-MODEL terminates and proved that it delivers sound results, by showing that, among other things, if the model contains no abstract states at the special program locations *memerror* and *asserterror* then the program really is error-free.

Rather than making our model extraction algorithm parametrised on a list of analysis modules, we presented it for a single module, performing module combination and cooperation at the module level: in Section 3.6 we provided a

module combinator $\diamond$ for *combining two modules into one*, which makes them cooperate by exchanging information. By iterated application of $\diamond$ we can cooperatively combine as many modules as we like.

In fact, we went further than planned in two ways:

Falsification We developed in Section 6.2, and formally proved the correctness of, a novel method for *falsifying* safety properties using the same kind of models, i.e. using only over-approximation. This falsification check may not be very precise but is so computationally inexpensive that we may consider it as "coming for free".

LTL safety checking In Section 6.1 we outlined (though did not formalise) an automaton-based procedure for model checking safety properties using a "two-level" fragment of LTL. By "two-level" we mean that the "propositions" of the temporal logic can themselves be any formulae of our first order logic $\mathscr{L}$. This model checking procedure may be slow and imprecise, but on the other hand it is very flexible: a huge range of ad-hoc queries can be made without having to recompute the model.

7.1.2 Revisiting our implementation objectives

I1 *Implement the central broker which coordinates the verification process, using the algorithm from D4.*

We programmed HECTOR, a prototype of the verification framework we designed, using Prolog as described in Sections 4.1 and 4.2. We chose Prolog because it has several features well-suited to rapid prototyping. Our implementation stores programs, configurations and models very simply, using dynamic predicates in the Prolog database (via `assert` and `retract`), and represents formulae of $\mathscr{L}$ with Prolog terms. Using SWI Prolog's module system, we place the code for each *analysis* module in a separate *Prolog* module, for clean

separation.

I2 *Implement an analysis module for predicate abstraction, which supports both the trivector and monomial variants.*

Section 4.4 details our analysis modules for these two predicate abstraction techniques. The two modules share most of their code. The existing theorem prover Simplify is used currently, but is accessed through a well-defined interface so the modules could be modified easily to use another prover.

Our modules use so-called *connecting formulae* rather than a predicate transformer (weakest precondition or strongest postcondition), as described in Subsection 4.4.1. This is because any shared formulae received are over several time indices, and we couldn't immediately see how to use these alongside a predicate transformer, which produces formulae over a single time index only. Procedure returns are dealt with by using special time indices $\mathbf{2}$ and $\mathbf{3}$.

We also cache the results of theorem prover calls, to avoid unnecessary repeated computation. This is especially important because when another module causes branching, the predicate abstraction modules often make identical theorem prover calls in each of the resulting branches.

I3 *Develop a module for three-valued shape analysis, by appropriately wrapping the TVLA software.*

Section 4.5 describes the implementation of our shape analysis module, using a slightly modified version of TVLA. This module is by far the most complicated of those we implemented, and the most configurable. Configuration options include: which variables/fields are tracked and which are ignored, which procedures are analysed and which are ignored, which variable are "focused", the use of instrumentation predicates (for reachability and cyclicity), and what patterns of formulae are shared.

To make use of shared formulae, our module translates them into TVLA's internal logic and then uses the coercion operation (Subsection 4.5.4). To provide

shared formulae, the module reuses this translation along with a procedure for generating candidate shared formulae, according to the general scheme given in Subsection 4.5.5.

While our shape analysis module worked well in our case study, we do see two potential areas of improvement.

- **More systematic translation:**

  As noted in Subsection 4.5.4, we translate literals from $\mathscr{L}$ into TVLA's internal logic by recognising a long list of particular cases, rather than by a systematic process. We do this because the systematic translation we tried produced formulae which, while correct, gave imprecise results under compositional three-valued semantics.

  We could in future investigate whether a translation exists which is systematic yet still adequately precise and efficient, perhaps by using patterns for minimisation as in [AH06].

- **Using a "free list":**

  In our module, the universe of the TVLA heaps is the set of allocated heap objects, rather than $\mathbb{Z}$. In retrospect this was a mistake. A better approach would be to use $\mathbb{Z}$ as the universe, modelling the unallocated addresses explicitly using a "free list", having an explicit node for null, and using a summary node to represent all negative values. Allocating a heap object would then correspond to removing an element from the free list. This approach would have several advantages:

  1. We could drop the restriction that quantifiers in $\mathscr{L}$ can only be translated if they are appropriately guarded by an allocation predicate.
  2. Our shape analysis module would never have to "give up" (as described in Subsection 4.5.2) because all possible values of variables could be modelled.

3. We could represent the restriction that a pointer is not null (due to the use of an explicit null node).

4. We would have the option in future of using the finite differencing methods developed in [RSL03], which would assist in introducing new forms of instrumentation. These methods cannot be used with statements, such as our `New` statement, which change the universe of the abstract heaps[1].

I4 *Provide an interface by which users can enter programs, configure and start the verification process, and monitor the results.*

Users of HECTOR provide input programs and configurations as text files, using Prolog syntax, as described in Section 4.2; models are also saved in this way. Such text files remain human readable and editable, and can be processed by many utilities; this is the UNIX tradition. Programs are entered as control flow graphs rather than as source code, which keeps the system simple, but eventually turned out to be a nuisance; this is discussed in the Future Directions Section 7.3.

In order that the user can explore models of programs, and understand counterexamples returned, HECTOR can produce various forms of graphical output, as we saw in Section 4.9. For ease of use, these visualisation functions are accessed through a web browser, where the models built are arranged into a tree of folders. Through this web interface, the outputs can be customised in various ways, for example by hiding the components from a particular analysis module. HECTOR can also show where sharing between modules proved useful.

HECTOR can automatically generate verification summaries of the kind seen in Figures 5.5 and 5.6, which give statistics about the use of sharing and about the configuration options used.

Finally, as described in Section 6.1, we allow interesting execution paths to be

[1] We thank Tal Lev-Ami for a helpful discussion of these issues.

found by queries in a "two-level" variant of safety LTL, where "propositions" can be arbitrary first order formulae (of $\mathscr{L}$). Even though such queries are not checked particularly quickly, finding interesting executions in this way is still much faster than manual inspection of the model.

I5 *Implement some shallow analysis modules to help out the sophisticated ones.*

We implemented four shallow analysis modules:

types (Section 4.6) places a type system on top of our untyped language, performing type inference to discover variables that stay within particular subsets of $\mathbb{Z}$. These subsets (types) are: non-negative, allocated, Boolean and null.

symbprop (Section 4.7) provides symbolic constant propagation.

compsigns (Section 4.3) provides a simple sign analysis. Formulae received from other modules are interpreted using a compositional three-valued semantics.

refs (Section 4.7) provides simple tracking of heap references: each tracked variable is classified as either 'null', 'ref' (a valid heap reference) or 'other'. The implementation, including treatment of incoming formulae, is similar to **compsigns**.

7.1.3 Revisiting our experimental objectives

E1 *The program verified is moderately sized, is based on a piece of real-world software, and uses diverse features.*

For our case study we verified an implementation of the MineSweeper puzzle game (detailed in Section 5.2). We adapted the (publicly available) program used for the case study in [LR07], which seemed a reasonable approach since that program uses all the main features on which we wanted to test HECTOR:

linked data structures, pointer arithmetic and recursion. Our adapted program consists of 356 program statements across 24 procedures together with 48 associated declarations (for fields, procedures, and shorthand formulae). As described in Subsection 5.2.3 we annotated the program with high-level properties, similar to those mentioned in [LR07], which make sense at the application level, but also with low-level assertions at the entry and return points of most procedures. This gave a total of 43 conditions to verify.

E2 *The verification employs (at least) two sophisticated domains, which are significantly different.*

We verified the MineSweeper program using modules for predicate abstraction and three-valued shape analysis. The two are significantly different: predicate abstraction is based on theorem proving, and can reason about numerical properties but not transitive closure, whereas three-valued shape analysis is based on models, and can reason about transitive closure but not numerical properties.

E3 *There is a* two-way *exchange of information between the sophisticated domains: each contributes information which the other uses to make its own analysis better in interesting ways.*

In Section 5.5 we traced through the model extraction algorithm following a particular "scenario" of MineSweeper gameplay, and pick out interesting instances of formula sharing. We see the predicate abstraction module helping the shape analysis module, and vice versa. The tables in Figure 5.6 confirm this, and provide simple statistics about the use of shared formulae.

E4 *One or more additional domains implementing shallow analyses are shown to help out the sophisticated domains, by handling easy cases or supplying information that can be easily obtained. The verification should run* faster *when these additional domains are used.*

When we used our four shallow modules alongside our predicate abstraction and shape analysis modules, our verification ran faster, and needed less user guidance. Specifically, the model extraction time was cut by 64 seconds, a reduction of 14%. 101 fewer configuration items were needed, a reduction of 32%. This was discussed in Section 5.4.

E5 *The domains used should implement techniques that are useful for software verification in general, and not be developed specifically just to make the case study work.*

Five of the analysis modules used in our case study implement generic, existing program analysis/abstraction techniques. The sixth, **refs**, which we invented ourselves, is uncomplicated and is generally applicable.

7.1.4 Revisiting the proposed benefits of our approach

The conjectured benefits B1-B4 of our approach, set out in Chapter 1, represent the long-term, high-level goals for the idea of modular domain combination, and are perforce ambitious and somewhat nebulous. Hence, we can only draw tentative conclusions here. Nevertheless, we dutifully revisit them one by one (again in slightly abbreviated form).

B1 Verifying programs with diverse features: *Abstract domains in use target specific aspects of program behaviour, such as numerical relationships, linked data structures, use of string buffers etc. But in applications programs, these features are all mixed up together. Verification systems based on a single specialised domain cannot verify such programs. By combining the different domains cooperatively, we hoped our system would solve a broader range of verification problems.*

We have successfully demonstrated this benefit in our case study. Our implementation of MineSweeper cannot be verified by first order predicate abstrac-

tion, because first order logic cannot even express the required heap reachability properties. Similarly the program cannot be verified by TVLA because it uses pointer arithmetic which TVLA cannot reason about. By combining the two approaches, and making them cooperate, we were able to verify the program.

B2 Using cheap analyses where applicable reduces workload: *We suspected that by using cheap but shallow analyses alongside more expensive but deeper ones, we could reduce the workload of the more expensive analyses. We hoped to gain the depth of the more expensive analyses, but with less computation.*

We have successfully demonstrated this benefit with our case study too. Using our shallow modules alongside our sophisticated ones reduced both the model extraction time (cut by 64 seconds, a reduction of 14%) and the amount of user guidance needed (101 fewer configuration items were needed, a reduction of 32%.)

B3 Implementation is more manageable: *The implementor of an analysis module just needs to implement the module interface: he only has to make his module "understand" the single common logic, and the new module will then cooperate with existing ones. The implementor does not need to know about the abstract constraints used internally by other modules. Thus, we can develop the verification system in small, easy to understand parts.*

Our personal experience of working with HECTOR is that this benefit is indeed provided. When we programmed our **tvla** module, for instance, we genuinely didn't have to worry about pointer arithmetic, and when we wrote our predicate abstraction modules, we genuinely didn't have to add mechanisms for proper transitive closure reasoning. Adding the shallow modules was especially easy.

In a sense, however, it is difficult for us to answer this question, because we understand the whole HECTOR system in any case. We suggest the following

experiment: ask a third party to implement a module for e.g. separation logic based shape analysis. This third party should be familiar with the common logic $\mathscr{L}$ and the interface for analysis modules, but not the rest of the HECTOR system. If such a third party can comfortably produce a working module, which can be "dropped in" to our case study in place of the **tvla** module, this is a good sign that our system is well structured.

B4 Abstractions can be mixed and matched, with the most appropriate chosen for each task: *Different verification tasks require or benefit from different kinds of abstractions. In our scheme, once abstractions have been suitably wrapped as modules, they can be "mixed and matched" freely, so we can select whatever kinds we need for each task.*

Because we have so far only undertaken one serious case study, there is little we can say about this proposed benefit. To investigate further, one could implement more domains, and have a group of users verify a broad range of programs. One could then see whether the users really did select different domains for the different verifications tasks, or whether they used the same few over and over; in the latter case, these might be the "best" domains.

7.2 Comparison with other approaches to domain combination

In Chapter 2 we described the classical *reduced product* operator which gives ideal domain combination, but is non-algorithmic, meaning that no modular implementation of the operator is possible. We also described work on the *open product* operator, which was the only research we could find that began to address the problem of combining domains in a modular way.

In this section we focus on recent work in this area. We compare our work to

some recently developed systems which perform (some kind of) automatic domain combination. These are systems which emerged either around the time we began our work on HECTOR in late 2004, or later. We cannot exhaustively discuss all related work in detail; among the interesting and relevant research not examined here is [BHT07]. We feel encouraged by the level of attention that this topic has received recently.

7.2.1 Comparison with Nelson-Oppen style systems

Various verification systems [GT06, CL05, RBHC07] have been built or proposed which base their domain combination on the well-known Nelson-Oppen technique [NO79], which is a way to combine reasoners for various *logical theories* into a reasoner for the combined theory:

> "In this paper we give a general method for combining decision procedures for two quantifier-free theories into a single decision procedure for their combination, which contains the functions and predicates of both theories. The method is based on a technique which we call equality propagation." [NO79]

For instance, in [NO79] it is shown how to combine decision procedures for various theories including: equality, integer arithmetic with $+$, $-$ and $\leq$, the Lisp functions for lists (*car*, *cdr* and *cons*), the theory of arrays (using functions *store* and *select*), and the theory of uninterpreted functions.

The Nelson-Oppen approach is similar to ours in that it allows reasoners for each of the theories to be developed separately, with a clearly defined interface, and then combined modularly. For example, while writing the decision procedure for arrays, one does not have to worry about terms containing $+$ and $-$; similarly, while writing

the decision procedure for linear arithmetic, one does not have to worry about array accesses.

Overall, however, the two approaches are not very alike. Here are some of the differences between them:

- In Nelson-Oppen, the class of constraints in each reasoner can only be a quantifier-free first order logical theory. In contrast, in our approach we can use any kind of constraints for which the appropriate functions (such as concretisation, γ) can be defined. Hence, for instance, TVLA models can be integrated within our approach but cannot be used with Nelson-Oppen.

- With Nelson-Oppen only *equalities between variables* are propagated between the reasoners being combined. In our approach, on the other hand, any formula of $\mathscr{L}$ can be propagated, including formulae which use quantifiers and transitive closure.

- The scheduling of propagation is different in the two approaches. In both, the propagation of a formula can allow the receiving reasoner/module to conclude further properties of interest. In our approach we do not (yet) try to take advantage of this; we simply perform one round of formula propagation for each successor computation. In Nelson-Oppen however, any additional equalities which become available are themselves propagated as soon as possible, and may then trigger further propagations in turn.

- In Nelson-Oppen the various theories which are to be combined must be disjoint, i.e. they must have no predicates or functions in common. This means that, for example, one cannot combine two reasoners which both produce conclusions about arithmetic inequalities. In our approach we can do this, and we do, such as when combining our **types** module with our **tpa** module. All our analysis modules accept the whole of $\mathscr{L}$, and it is up to each module to decide which information in a particular formula it finds useful.

Much is made of the fact that the Nelson-Oppen approach provides the most precise possible combination of the decision procedures being combined. For example, [RBHC07] cites our work with the following comment:

> "Similarly, Charlton and Huth propose a software model checker in which separate analysis plugins (such as for heaps and for other theories) can cooperate, but the communication is ad hoc, so there are no guarantees that all interactions between theories are propagated."

We would like to point out, however, that results about the completeness or maximal precision of the Nelson-Oppen technique are only obtained by placing very strong restrictions on the theories being combined. In [RBHC07] the condition given is that the theories are stably infinite quantifier-free first order theories with disjoint signatures.

We have not pursued such results for our framework because we know that, for any reasonably general definition of an abstraction domain, maximally precise combination will be impossible. To see this, note that, using only our predicate abstraction module **tpa**, computing successors for a single `Skip` statement is already an undecidable problem.

We now mention some specific Nelson-Oppen style systems (to which the above general comments apply):

- [RBHC07] describes an SMT (satisfiability modulo theories) solver called MathSAT, in the spirit of Nelson-Oppen, which includes a theory for some particular forms of heap reachability. This theory can express the reachability of one variable from another via a particular pointer field, and a related notion of "betweenness". MathSAT is then used for predicate abstraction of heap-manipulating programs, combining reachability and data value reasoning.

- [CL05] provides a *congruence-closure domain*, which is parametrised by a set of *base domains* and effects a weak combination of those domains. The method is again related to the Nelson-Oppen technique. A base domain is given which is meant to track the heap, and allow other domains to treat heap expressions as if they were simple variables. However this *heap succession domain* is based on the theory of arrays (with *select* and *store* operations) and so cannot deal with heap properties such as reachability or cyclicity.

- [GT06] provides a *logical product* operator for a particular class of domains, the *logical lattices*. This method is very close to Nelson-Oppen, and hence propagates from one domain to another only equalities between variables. The theories considered in this paper are those of linear arithmetic, parity, sign, uninterpreted functions and Lisp lists.

7.2.2 Comparison with the Hob system

The Hob system [KLZR05, LKR05] also allows different analysis techniques to co-operate in order to verify a program, and largely shares the goals of our work:

> "We have developed a variety of analyses, with each specific analysis structured to verify a specific, fairly narrow class of data structures. The ability to target each analysis to a specific class of data structures has provided substantial benefits. Eliminating the burden of building a single general analysis has reduced the overall development overhead and enabled us to produce very narrow but very sophisticated analyses with relatively little engineering effort. It has also reduced the amount of broad expertise any one person needs to acquire to develop an analysis." [KLZR05]

The mechanism by which domains interact in Hob is quite different, however. Target programs must be structured into modules, and exactly one domain is used to

analyse each of these. At the boundary of each source code module, the user defines some abstract *sets* of heap objects; the way these sets are defined is specific to the domain used. For example, for a source code module implementing a linked list, one might define an abstract set denoting all list elements. It is through these sets that the domains interact: BAPA (the theory of Boolean algebra with Presburger arithmetic [KNR05]) is used as a common logic, and also for behavioural assertions.

Thus, each program statement is analysed by a single domain, and interaction between domains only occurs at the boundaries of source code modules. This is an important difference between Hob and HECTOR — Hob's authors strive to "decouple" the analyses whereas we seek to integrate them more tightly.

7.2.3 Comparison with the Jahob system

The Jahob system [ZKR08] also combines different reasoning techniques, using them collaboratively to verify programs, but uses a different technique called *integrated reasoning*. The system is based on generating and proving verifications conditions; the logic used is full classical higher-order logic. The central idea, referred to as *splitting*, is to replace each complicated verification condition with a conjunction of simpler ones. Each simpler conjunction is then dispatched to an appropriate reasoner.

> "A typical data structure operation generates a verification condition that splitting separates into a few hundred implications, each of which is a candidate for any of the provers." [ZKR08]

This approach shares our goal of exploiting shallower, cheaper or more specialised analyses where possible: if the system recognises that a certain conjunct is of the kind that can be solved well by a particular prover, this prover can be invoked. If a conjunct cannot be proved by any of the automated provers, the user is required

to supply the proof using an interactive proof assistant. It is not clear what level of automation Jahob achieves.

We note that Jahob exploits existing first-order provers by translating a subset of higher-order logic into first-order logic [BKW+07]. We had conjectured that this could not be done effectively (see our discussion in Section 3.7) but it seems we were mistaken.

7.2.4 Comparison with the ASTRÉE system

[CCF+06] details the domain combination mechanism used in ASTRÉE [CCF+05], an industrial tool for verifying mission-critical embedded software. As our HECTOR program is a prototype written by a single person, it would be absurd to compare the overall effectiveness of the two, but we will comment on conceptual differences relevant to the domain combination techniques.

Like our approach, ASTRÉE features a modular product operator which approximates the reduced product. The interface for abstraction domains provides operations *EXTRACT*, which corresponds to our *share*, and *REFINE* which corresponds to our extra parameter to *succ*. The stated aims of this are similar to ours, e.g.

> "This modular design allows abstract domains to be turned on and off by runtime options, easy addition of new domains, and the suppression of older domains that have been superseded by newer ones. [...] ASTRÉE is therefore an extensible abstract interpreter." [CCF+06]

Here are some differences between the two approaches.

- The scheduling of information propagation in ASTRÉE is much more elaborate than in our approach. For instance, the product operator is not commutative,

with (say) the domain on the left being run first, and certain kinds of communication flowing from the left domain to the right domain but not vice versa. Thus when several domains are combined they are formed into a hierarchical structure.

- In ASTRÉE there is no single common language for propagation. Instead, there are many special types of constraint that can be communicated, and it is up to each domain to decide which types of constraints it can deal with.

  > "Constraints are implemented with a sum type, where each summand is a different kind of constraints. After each computation, each domain collects a list of constraints [...]. The primitive *REFINE* scans the list and refines the abstract properties accordingly." [CCF+06]

  It is not entirely clear from [CCF+06] what these different kinds of constraints, or "summands", are in practice, but it appears that they are simple numerical or Boolean relations between program variables (and thus quantifier-free).

- Abstract values in ASTRÉE represent sets of *traces* rather than sets of states.

- ASTRÉE doesn't handle dynamic memory allocation, and therefore doesn't handle linked data structures. The domains used in ASTRÉE appear to be mostly numerical in nature (such as intervals, octagons, ellipsoids, arithmetic-geometric progressions).

- Recursion is not supported in ASTRÉE.

- ASTRÉE provides some domain constructors other than the product operator, such as a *Boolean partitioning* domain transformer.

- ASTRÉE supports widening and uses it heavily.

7.3 Future directions

During the course of this work, we have become aware of far more new research questions than we care to list. Here, in no particular order, is a small sample.

1. How can we incorporate widening into our approach?
2. What should be the "most thorough" semantics for checking linear time temporal logic properties over our models?
3. How does the integration of a module based on X work out? (where X is ownership types, graph grammars, graph types, separation logic, octahedra etc.)
4. Given that only a small fraction of shared formulae turn out to be useful to another module, is there a way to avoid so much superfluous sharing without losing precision? (perhaps by means of a lazy sharing strategy, in the spirit of lazy abstraction [HJMS02])
5. What happens when feature X is added to the language? (where X is exceptions, inheritance, garbage collection etc.)
6. How does one do termination arguments in our module-based system?
7. Can our framework be extended to deal with multi-threaded programs by borrowing from existing work?
8. Do we really need to use connecting formulae instead of predicate transformers in our predicate abstraction modules?
9. If so, is there something special that theorem provers can do to better reason about connecting formulae (which follow a set pattern)?
10. Can we make our target language's semantics closer to those of "real" programming languages by using partial functions, and if so, how should we represent partial functions in the logic $\mathscr{L}$?

Also, of course, it would clearly be desirable to perform more, and more diverse, verification case studies with HECTOR to gain more insight into how applicable the approach is in practice.

Having said all that, we now outline in greater detail what we believe are the three most important directions for the continuation of this line of research (for reasons we will explain): one practical, one theoretical and one that is a blend of the two.

7.3.1 Practical issue: processing source code

As we have seen, the current version of HECTOR takes as input specially prepared, textually represented CFGs. While this was useful during the early prototyping period, because it kept the HECTOR system simple, the fact remains that if we want to verify existing programs rather than those created specifically to exercise the verifier, we must accept source code input, perhaps for a subset of C. Additionally, even if we are only interested in analysing specially prepared case study programs, programming by manually constructing control flow graphs is sufficiently tedious and error-prone to be a serious impediment. Therefore we conclude that to take HECTOR any further, one must add source code input features.

We noted in Section 2.1.1 that it is fairly easy to use a front-end or compiler to automatically produce CFGs from source code, and were content to leave it at that. In hindsight, however, that isn't quite the end of the story: the problem is that any counterexample trace found is a trace through the CFG, and not through the source code. The user has never seen the CFG, and in it, complex statements have been replaced with groups of simple statements, new variables have been introduced to temporarily hold intermediate values, and the alternation and repetition structures have been encoded with simple guarded edges. How, then, could we make counterexample traces intelligible to the user? We see three possible approaches.

1. **Expand the set of statements which can label CFG edges, including**

e.g. assignment statements with arbitrary expressions on the right hand side.

The advantage of this approach is that it would make the relationship between the source code and the CFGs much simpler; counterexample traces would become easier to follow and could be mapped back to the source code if desired.

Unfortunately there is a serious disadvantage: the abstract successor functions would have to deal with far more, and far trickier, forms of statement, requiring more, and more subtle, programming. This would hit HECTOR especially hard: whereas other tools include one set of abstract successor functions, HECTOR has *one set per analysis module*, so the extra cost would be multiplied.

Also, in this approach the semantics of the target programming language become more complicated to define and reason about, so that proving a particular analysis module sound would become more difficult. Again this increased difficulty is multiplied by the number of analysis modules.

Finally it is not clear how the introduction of more complex statement forms would affect the behaviour of formula sharing: since we currently perform one round of sharing per simple statement, it is conceivable that we may have to perform several rounds per complex statement to get the same outcome.

2. **Compile complex statements down to groups of simple statements, but hide this from the user.**

   With this approach we would compile the source code into CFGs replacing each complex statement with a group of simple statements, and introducing temporary variables, but we would never let the user see the CFGs. Instead we would map each abstract counterexample back onto a trace through the source program, and show this to the user.

   We are skeptical about whether this is possible. A recent paper [LF08] discusses the issue, finding that (in a particular formal setup) there *is no systematic way* to faithfully map between analysing at the source code level and

analysing at the level of the CFG. In particular, the behaviour of the analysis at CFG level really does depend on how the temporary variables are treated, and these variables are not visible at the source code level.

3. **Compile complex statements down to groups of simple statements, admit we are doing it and try to make it palatable.**

   In this approach, which we prefer, the user would be shown counterexample traces at CFG level, and would configure the analyses at CFG level, referring to temporary variables in the configuration where necessary. We would make CFG-level traces as easy to understand as possible, by annotating each group of CFG nodes with the source code statement that generated them. We would also make sure the compilation process was simple and predictable, because if the compiler performed any clever optimisation or rearrangement of the code, this would confuse the user. (We would want a simple compiler anyway, because the correctness of the verification would additionally depend on the correctness of the compilation.)

   Of the three, this approach requires the least work.

7.3.2 Practical and theoretical issue: adding CEGAR facilities

In Section 5.3 we described the abstract-check-fine loop, by which we obtained better and better configurations for the analyses, until we had verified the program. The process of improving the configuration to eliminate the current counterexample was performed by hand at each stage, which is a time-consuming and sometimes frustrating process and requires expertise. It is unsurprising, then, that in recent years much research has been done into methods of automating this process.

Such methods, known as *Counterexample-Guided Abstraction Refinement* or CEGAR [CGJ+00], perform a systematic scan of the abstract counterexample trace,

and then suggest a better configuration. CEGAR has been very successfully used for tools based on predicate abstraction (e.g. [BR02, HJMS02]), where it suggests new abstraction predicates; this is crucial in the success of Microsoft's Static Driver Verifier [BBC+06], because it enables the verifier to run fully automatically and thus places verification technology at the fingertips of non-expert users. CEGAR methods have also been developed for TVLA which suggest new instrumentation predicates [LRS05].

We believe that adding such features to HECTOR is an important goal for two reasons.

1. **CEGAR can make verifications much easier to perform.**

   A good implementation of CEGAR would allow HECTOR to verify programs with much less user guidance, so that verifying programs would become easier. This would open the door to building a moderate collection of verified programs, which in turn would allow us to gain more insight into how module-based verification works: we could search for common patterns of sharing, test the effect of new optimisations across a range of examples, etc..

2. **Module-based CEGAR is an interesting research problem in its own right.**

   The module-based design of HECTOR creates new issues for CEGAR which do not arise in single-domain systems.

   Firstly, because HECTOR runs several analyses together, how do we decide which one to reconfigure? For example, we may have the choice between adding a new predicate to the predicate abstraction module, or adding a new instrumentation predicate to the TVLA module.

   This is reminiscent of the two-level CEGAR algorithm in MAGIC [CCG+04], which decides whether to introduce more abstraction predicates, or refine its use of a technique called *action abstraction*. However, in MAGIC these two abstractions are layered one on top of the other, rather than side-by-side as

in our work. A similar issue is encountered in work on automatically proving termination [CPR05], where the system must choose whether to refine the choice of abstraction predicates, or refine the set of relations used to try to establish termination.

Secondly, is it possible to successfully perform CEGAR with a generic algorithm in the broker, or would specialised CEGAR routines be needed for each analysis module? In either case, what changes to the module interface are required? In terms of generic CEGAR, our interest is aroused by [GR06] which shows how to perform CEGAR on any abstract domain which uses a widening operator to make the analysis converge. The idea is that, for any spurious counterexample, one can pinpoint which application of widening is to blame, and then widen less aggressively at that point in the analysis. This method is appealingly simple but not directly applicable to our framework (because for a start we do not use widening).

7.3.3 Theoretical issue: generalising interpolation

We finish by outlining a broad theoretical question. Propositional logic has the following property: If Φ and Θ are formulae and $\neg(\Phi \wedge \Theta)$ is valid, then there exists a formula Ψ such that: $\Phi \rightarrow \Psi$ is valid, $\Psi \rightarrow \neg\Theta$ is valid and Ψ contains only propositional letters occurring in *both* Φ *and* Θ. This is a slight reformulation of the well-known Craig Interpolation Theorem. Analogous results hold for various other logics including first order logic (e.g. [CK90]). The formula Ψ is known as an interpolant because it lies between Φ and $\neg\Theta$ in the entailment ordering. In a sense, the above result says that, given two formulae Φ and Θ which are mutually inconsistent, the *reason for their inconsistency* can be expressed using only their common propositional letters. A wider question hinted at by such theorems is: given two mutually inconsistent formulae Φ and Θ, what "logical resources" are required to express the reason for their inconsistency?

Let us make a loose analogy between the above result and our module-based verification system. Suppose we have two analysis modules $\mathbf{M}$ and $\mathbf{N}$, and pick out two abstract values $a \in \mathbf{M}.T$ and $a' \in \mathbf{N}.T$. Suppose that these abstract values are mutually inconsistent in the sense that $\mathbf{M}.\gamma(a) \cap \mathbf{N}.\gamma(a') = \emptyset$. The question is then: what logical machinery is required to witness the inconsistency between a and a', i.e., for what logics $\mathscr{T}$ is there guaranteed to exist a formula $\Psi \in \mathscr{T}$ such that a entails Ψ, and Ψ "rules out" a'? The idea is that this Ψ would be the formula shared by the module $\mathbf{M}$, and that the other module $\mathbf{N}$ would then use Ψ to notice the inconsistency, and generate no successors.

A concrete instance of this question might be: if we have an arbitrary symbolic heap expressed in a fragment of separation logic, and an abstract TVLA heap (i.e. a three-valued heap model), and if the two are mutually inconsistent, is this fact witnessed by a quantifier-free TC formula?

In general we would like to know, for any particular combination of modules, what sort of common logic $\mathscr{T}$ will be sufficient to allow the detection of inconsistency where it exists. If $\mathscr{T}$ is quite a "small" logic, for instance, this may allow us to restrict our attention to formulae of $\mathscr{T}$ when propagating information. We are unaware of any existing work in this direction.

7.4 Closing remarks

Sadly, we have reached the end of our journey through the world of cooperatively combining program verifiers, which began in Chapter 1 when, dissatisfied with the limited applicability of the specialised program verification and analysis systems in current use, we decided to go in search of a way to combine these specialised systems modularly in order to exploit the advantages of each.

Before setting out, we surveyed the surrounding landscape: in Chapter 2 we ex-

amined existing abstraction methods, noting their strengths and weaknesses, and came across the open product operator which had been used to combine domains for optimisation of logic programs.

The first stage of our journey proper, Chapter 3, took us into uncharted territory. We designed a modular verification framework, based on the concept of analysis modules, and formalised all aspects of it. In the next leg, Chapter 4, we implemented the framework to a very useable level in our software model checker HECTOR.

Chapter 5, in which we conducted a case study, using HECTOR to verify an implementation of the puzzle game MineSweeper, confirmed that our efforts so far had been very worthwhile: combining domains in a modular way was shown to extend the range of programs that can be verified, and increase the ease and speed of verification. In Chapter 6 we pressed on a little further in our travels, before stopping to reflect in this Chapter on how far we have come.

Throughout this thesis, we have aimed to share with the reader not just the main points of our work, but also the little nuggets of information we have learned along the way. It is our sincere hope that our results, and the open questions we have identified, will help justify and motivate further research on (some form of) automatic modular combination of abstraction domains.

Appendix A

A.1 Three-valued logic

The idea of three-valued logic, as in [Kle52], is to use an extra truth value *unknown* in addition to the usual *true* and *false*. Intuitively *unknown* is used to represent lack of knowledge about a particular aspect of a system.

Thus, three-valued logic accounts very naturally for the fact that the process of abstraction gives rise to only partial information, and allows us to make the most of the information we do have: even if a formula Φ contains a subformula which evaluates to *unknown*, we may still be able to obtain a definite truth value for Φ.

For example, in propositional three-valued logic, the propositional letters $p, q, r, \ldots$ are assigned truth values from $\{true, false, unknown\}$. The three-valued semantics of each connective is given in Figure A.1. Consider the formula

$$p \vee (q \wedge r)$$

under the truth assignment

$$\{p \mapsto true, q \mapsto true, r \mapsto unknown\}$$

The subformula $q \wedge r$ evaluates to *unknown* because the truth of r is unknown;

A	$\neg A$
true	false
unknown	unknown
false	true

A	B	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
true	true	true	true	true	true
true	unknown	unknown	true	unknown	unknown
true	false	false	true	false	false
unknown	true	unknown	true	true	unknown
unknown	unknown	unknown	unknown	unknown	unknown
unknown	false	false	unknown	unknown	unknown
false	true	false	true	true	false
false	unknown	false	unknown	true	unknown
false	false	false	false	true	true

Figure A.1: Three-valued compositional semantics for logical connectives.

nevertheless, the formula as a whole evaluates to *true*.

Note that the same three-valued semantics for connectives emerges if one lifts the two-valued connectives to the level of sets of truth values, applying them pointwise, and reads $\{true, false\}$ as *unknown*.

Three-valued semantics have also been given to temporal logics (e.g. [HJS01]) and first-order predicate logics [NNS01]. The TVLA system discussed in Subsection 2.5.3 (page 65 onwards) uses models of three-valued predicate logic to represent sets of possible program heaps; another use of three-valued logic is described in [GHJ01].

A.1.1 Compositional semantics vs. thorough semantics

A simple structural recursion suffices to evaluate formulae according to the semantics of Figure A.1, and for this reason those semantics are called *compositional*. Unfortunately however, the compositional semantics do not always behave as we

would like. For example, evaluating the formula

$$(p \vee \neg p) \wedge q$$

under the truth assignment $A := \{p \mapsto \mathit{unknown}, q \mapsto \mathit{true}\}$ produces the result *unknown*, yet in every two-valued truth assignment consistent with A (namely $\{p \mapsto \mathit{true}, q \mapsto \mathit{true}\}$ and $\{p \mapsto \mathit{false}, q \mapsto \mathit{true}\}$) the formula evaluates to *true*.

Intuitively this is because the compositional semantics consider that p might be true in one disjunct, and simultaneously false in another, which cannot really happen, of course. This effect is not confined to formulae containing a tautology.

Note that even though the formulae $(p \vee \neg p) \wedge q$ and q are equivalent in two-valued logic, they behave differently under the compositional three-valued semantics; q always gives a more precise result than $(p \vee \neg p) \wedge q$.

This leads to the definition of a *thorough* or *supervaluational* semantics [vF69], which evaluates the formula in all two-valued truth assignments that are consistent with the three-valued one; if the results of this are all *true* (resp. all *false*) then the thorough semantics produces *true* (resp. *false*), and *unknown* otherwise.

The thorough semantics gives answers which are at least as good as, and may be more precise than, those given by the compositional semantics, but thorough checking is usually more computationally expensive than compositional checking.

Works comparing the compositional and thorough semantics include [RLS02] for propositional logic, [AH06] for CTL $\cap$ LTL and [BG00, GH05] for other temporal logics.

A.2 Tables for sign analysis

x	y	$x = y$	$x < y$	$x \leq y$
pos	pos	unknown	unknown	unknown
pos	zero	false	false	false
pos	neg	false	false	false
pos	any	unknown	unknown	unknown
zero	pos	false	true	true
zero	zero	true	false	true
zero	neg	false	false	false
zero	any	unknown	unknown	unknown
neg	pos	false	true	true
neg	zero	false	true	true
neg	neg	unknown	unknown	unknown
neg	any	unknown	unknown	unknown
any	pos	unknown	unknown	unknown
any	zero	unknown	unknown	unknown
any	neg	unknown	unknown	unknown
any	any	unknown	unknown	unknown

x	y	$x+y$	$x-y$	$x \times y$
pos	pos	pos	any	pos
pos	zero	pos	pos	zero
pos	neg	any	pos	neg
pos	any	any	any	any
zero	pos	pos	neg	zero
zero	zero	zero	zero	zero
zero	neg	neg	pos	zero
zero	any	any	any	zero
neg	pos	any	neg	neg
neg	zero	neg	neg	zero
neg	neg	neg	any	pos
neg	any	any	any	any
any	pos	any	any	any
any	zero	any	any	zero
any	neg	any	any	any
any	any	any	any	any

x	$allocd_{\mathbf{i}}(x)$
pos	unknown
zero	false
neg	false
any	unknown

A.3 Full soundness conditions for analysis modules

sound-m-init There exists $a \in \mathit{init}(\cdot)$ such that $(s^{\mathit{start}}, s^{\mathit{start}}) \in \gamma(a)$.

sound-share-skip If

1. $\pi, s_0 \xrightarrow{l,s} l', s$
2. $(s_0, s) \in \gamma(a)$

then $(s_0, s, s) \in [\![\mathit{share}(a, \texttt{Skip}, \pi)]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

sound-succ-skip If

1. $\pi, s_0 \xrightarrow{l,s} l', s$
2. $(s_0, s) \in \gamma(a)$
3. $(s_0, s, s) \in [\![\Phi]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

then there exists $a' \in \mathit{succ}(a, \texttt{Skip}, \pi, \Phi)$ such that $(s_0, s) \in \gamma(a')$.

sound-share-varcopy If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A)$
5. $e' = e \oplus \{u \mapsto e(v)\}$

then $(s_0, s, s') \in [\![\mathit{share}(a, \texttt{VarCopy}(u, v), \pi)]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

sound-succ-varcopy If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$

3. $s = (e, h, A)$
4. $s' = (e', h, A)$
5. $e' = e \oplus \{u \mapsto e(v)\}$
6. $(s_0, s, s') \in [\![\Phi]\!]^{\{\mathbf{O},\mathbf{I},\mathbf{C}\}}$

then there exists $a' \in succ(a, \texttt{VarCopy}(u, v), \pi, \Phi)$ such that $(s_0, s') \in \gamma(a')$.

sound-share-assignconst If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A)$
5. $e' = e \oplus \{u \mapsto k\}$

then $(s_0, s, s') \in [\![share(a, \texttt{AssignConst}(u, k), \pi)]\!]^{\{\mathbf{O},\mathbf{I},\mathbf{C}\}}$

sound-succ-assignconst If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A)$
5. $e' = e \oplus \{u \mapsto k\}$
6. $(s_0, s, s') \in [\![\Phi]\!]^{\{\mathbf{O},\mathbf{I},\mathbf{C}\}}$

then there exists $a' \in succ(a, \texttt{AssignConst}(u, k), \pi, \Phi)$ such that $(s_0, s') \in \gamma(a')$.

sound-share-arith If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$

2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A)$
5. $e' = e \oplus \{u \mapsto e(v_1) \otimes e(v_2)\}$

then $(s_0, s, s') \in [\![\mathit{share}(a, \texttt{Arith}(u, v_1, \otimes, v_2), \pi)]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

sound-succ-arith If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A)$
5. $e' = e \oplus \{u \mapsto e(v_1) \otimes e(v_2)\}$
6. $(s_0, s, s') \in [\![\Phi]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

then there exists $a' \in \mathit{succ}(a, \texttt{Arith}(u, v_1, \otimes, v_2), \pi, \Phi)$ such that $(s_0, s') \in \gamma(a')$.

sound-share-fieldread If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A)$
5. $e(v) \in A$
6. $e' = e \oplus \{u \mapsto h(f, e(v))\}$

then $(s_0, s, s') \in [\![\mathit{share}(a, \texttt{FieldRead}(u, v, f), \pi)]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

sound-succ-fieldread If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$

2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A)$
5. $e(v) \in A$
6. $e' = e \oplus \{u \mapsto h(f, e(v))\}$
7. $(s_0, s, s') \in [\![\Phi]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

then there exists $a' \in \mathit{succ}(a, \texttt{FieldRead}(u, v, f), \pi, \Phi)$ such that $(s_0, s') \in \gamma(a')$.

sound-share-fieldwrite If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e, h', A)$
5. $e(v) \in A$
6. $h' = h \oplus \{(f, e(v)) \mapsto e(u)\}$

then $(s_0, s, s') \in [\![\mathit{share}(a, \texttt{FieldWrite}(v, f, u), \pi)]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

sound-succ-fieldwrite If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e, h', A)$
5. $e(v) \in A$
6. $h' = h \oplus \{(f, e(v)) \mapsto e(u)\}$
7. $(s_0, s, s') \in [\![\Phi]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

then there exists $a' \in \mathit{succ}(a, \texttt{FieldWrite}(v, f, u), \pi, \Phi)$ such that $(s_0, s') \in \gamma(a')$.

sound-share-new If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A')$
5. $e(v) > 0$
6. $a > 0$
7. $\{a, a+1, \ldots, a+e(v)-1\} \cap A = \emptyset$
8. $A' = A \cup \{a, a+1, \ldots, a+e(v)-1\}$
9. $e' = e \oplus \{u \mapsto a\}$

then $(s_0, s, s') \in [\![\mathit{share}(a, \texttt{New}(u, v), \pi)]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

sound-succ-new If

1. $\pi, s_0 \xrightarrow{l,s} l', s'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s' = (e', h, A')$
5. $e(v) > 0$
6. $a > 0$
7. $\{a, a+1, \ldots, a+e(v)-1\} \cap A = \emptyset$
8. $A' = A \cup \{a, a+1, \ldots, a+e(v)-1\}$
9. $e' = e \oplus \{u \mapsto a\}$
10. $(s_0, s, s') \in [\![\Phi]\!]^{\{\mathfrak{O},\mathfrak{I},\mathfrak{C}\}}$

then there exists $a' \in succ(a, \texttt{New}(u, v), \pi, \Phi)$ such that $(s_0, s') \in \gamma(a')$.

sound-succ-call If

1. $\pi, s_0 \rightarrow l, s : \pi', s_0'$
2. $(s_0, s) \in \gamma(a)$
3. $s = (e, h, A)$
4. $s_0' = (e', h, A)$
5. $e'(x) = \begin{cases} e(p_i) & \text{if } x \text{ is } f_i \\ 0 & \text{otherwise} \end{cases}$ where $[f_1, \ldots, f_j] = Formals(\pi')$

then there exists $a' \in succ_C(a, \pi, \pi', [p_1, \ldots, p_k])$ such that $(s_0', s_0') \in \gamma(a')$.

sound-succ-return If

1. $\pi, s_0 \rightarrow l_1, s_1 : \pi', s_2$
2. $\pi', s_2 \xrightarrow{p'} l_3, s_3$
3. $(s_0, s_1) \in \gamma(a)$
4. $(s_2, s_3) \in \gamma(a')$
5. $Edges(Graph(\pi))(l_1) = \texttt{Call}(u, \pi', [p_1, \ldots, p_k]) : l$
6. $Edges(Graph(\pi'))(l_3) = \texttt{Return}(v)$
7. $s_1 = (e_1, h_1, A_1)$
8. $s_2 = (e_2, h_1, A_1)$
9. $s_3 = (e_3, h_3, A_3)$
10. $s = (e, h_3, A_3)$
11. $e_2(x) = \begin{cases} e_1(p_i) & \text{if } x \text{ is } f_i \\ 0 & \text{otherwise} \end{cases}$ where $[f_1, \ldots, f_j] = Formals(\pi')$
12. $e = e_1 \oplus \{u \mapsto e_3(v)\}$

then there exists $a'' \in succ_R(a, a', \pi, \pi', u, v, [p_1, \ldots, p_k])$ such that $(s_0, s) \in \gamma(a')$.

Bibliography

[AH06] Adam Antonik and Michael Huth. Efficient Patterns for Model Checking Partial State Spaces in CTL ∩ LTL. *Electronic Notes in Theoretical Computer Science*, 158:41–57, May 2006. Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics; 24-27 May 2006, Genova, Italy.

[Apt81] Krzysztof Apt. Ten years of Hoare's logic: A survey—part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[Bac88] Ralph-Johan Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV (Computer Aided Verification)*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer-Verlag, July 2007.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[BCO04] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS 2004 (Foundations of Software Technology and Theoretical Computer Science)*, volume 3328 of *LNCS*, pages 97–109. Springer, December 2004.

[BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO (Formal Methods for Components and Objects)*, pages 115–137, 2005.

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS 2005*, volume 3780 of *LNCS*, pages 52–68. Springer-Verlag, 2005.

[BG00] Glenn Bruns and Patrice Godefroid. Generalized model checking: Reasoning about partial state spaces. *Lecture Notes in Computer Science*, 1877:168+, 2000.

[BG03] Doron Bustan and Orna Grumberg. Simulation-based minimization. *ACM Trans. Comput. Logic*, 4(2):181–206, 2003.

[BHMV05] Ahmed Bouajjani, Peter Habermehl, Pierre Moro, and Tomas Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *TACAS*, pages 13–29, 2005.

[BHRZ03] Roberto Bagnara, Patricia Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. In *SAS (Static Analysis Symposium)*, pages 337–354, 2003.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *CAV*, pages 504–518, 2007.

[BKW+07] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin C. Rinard. Using first-order theorem provers in the Jahob data structure verification system. In *VMCAI*, pages 74–88, 2007.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. *The Spec# Programming System: An Overview*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, January 2005.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Lecture Notes in Computer Science*, 2031:268+, 2001.

[BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Proceedings of SPIN 2000*, volume 1885 of *LNCS*, pages 113–130. Springer Verlag, 2000.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001: Proceedings of the 8th international SPIN workshop on model checking of software*, volume 2057 of *LNCS*, pages 103–122. Springer Verlag, 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical report, Microsoft, 2002. MSR-TR-2002-09.

[BTSR04] Lars Birkedal, Noah Torp-Smith, and John Reynolds. Local reasoning about a copying garbage collector. In *POPL 04 (Principles of Programming Languages)*, pages 220–231, 2004.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130, Paris, France, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 77 (Principles of Programming Languages)*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL 79 (Princples of Programming Languages)*, pages 269–282, 1979.

[CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, London, UK, 1992. Springer-Verlag.

[CC04] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In *SAS (Static Analysis Symposium)*, pages 312–327, 2004.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREE analyzer. In *ESOP*, pages 21–30, 2005.

[CCF+06] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the ASTREE static analyzer. In *ASIAN*, pages 272–300, 2006.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[CCG+04] Sagar Chaki, Edmund M. Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.

[CCH00] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming: Open product and generic pattern construction. *Science of Computer Programming*, 38(1-3):27–71, August 2000.

[CDD+08] David Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander Summers. Universe types for

topology and encapsulation, 2008. To appear in post-proceedings of *FMCO 07 (Formal Methods for Components and Objects)*.

[CDE07] David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe types for race safety. In *VAMP 07 (Verification and Analysis of Multi-threaded Java-like Programs)*, pages 20–51, September 2007.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[CGL92] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *POPL 92 (Principles of Programming Languages)*, pages 343–354, New York, 1992. ACM.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL 78 (Principles of Programming Languages)*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York.

[CH07] Nathaniel Charlton and Michael Huth. Hector: software model checking with cooperating analysis plugins. In *Proc. International Conference on Computer-Aided Verification (CAV 2007)*, volume 4590 of *LNCS*. Springer, 2007.

[CH08] Nathaniel Charlton and Michael Huth. Falsifying safety properties through games on over-approximating models, 2008. In proceedings of Workshop on Reachability Problems (WRP) 2008.

[Cha06a] Nathaniel Charlton. Program verification with interacting analysis plugins. *Formal Aspects of Computing*, 2006. Springer Verlag, DOI: 10.1007/s00165-007-0029-4.

[Cha06b] Nathaniel Charlton. Program verification with interacting analysis plugins. Technical Report 2006/11, Department of Computing, Imperial College London, September 2006. ISSN 1469-4174.

[Cha06c] Nathaniel Charlton. Verification of Java programs with interacting analysis plugins. *Electronic Notes in Theoretical Computer Science*, 145, Proceedings of the 5th International Workshop on Automated Verification of Critical Systems (AVoCS 2005):131–150, 2006.

[CK90] Chen Chung Chang and Jerome Keisler. *Model Theory (3rd ed.)*. Elsevier, 1990. ISBN 978-0-444-88054-3.

[CL05] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI'05*, volume 3385 of *LNCS*, pages 147–163. Springer Verlag, 2005.

[CPR05] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction-refinement for termination. In *SAS (Static Analysis Symposium)*, volume 3672 of *Lecture Notes in Computer Science*, page 15, London, UK, September 2005. Springer.

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and Tony Hoare. *Structured programming*. Academic Press Ltd., London, UK, 1972. ISBN 0122005503.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, volume 1633 of *LNCS*, pages 160–171, 1999.

[Dij70] Edsger W. Dijkstra. Notes on Structured Programming. Available as EWD249 from Dijkstra Archive, April 1970.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[Dis03] Dino Distefano. *On model checking the dynamics of object-based software: a foundational approach*. PhD thesis, University of Twente, 2003.

[DM05] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[DN03] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 310–324, London, UK, 2003. Springer-Verlag.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag.

[FM97] Pascal Fradet and Daniel Le Métayer. Shape types. In *POPL 97 (Principles of Programming Languages)*, pages 27–39, New York, 1997. ACM Press.

[FS00] Alain Finkel and Grégoire Sutre. Decidability of reachability problems for classes of two counters automata. *Lecture Notes in Computer Science*, 1770:346+, 2000.

[GC06] Arie Gurfinkel and Marsha Chechik. Why waste a perfectly good abstraction? In *TACAS*, pages 212–226, 2006.

[GH05] Patrice Godefroid and Michael Huth. Model checking vs. generalized model checking: semantic minimization for temporal logics. In *Twentieth Annual IEEE Symposium on Logic in Computer Science*, pages 158–167, June 2005.

[GHJ01] Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR 2001 - concurrency theory: 12th international conference, Aalborg, Denmark, 20 - 25 August 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440, 2001.

[GJ02] Patrice Godefroid and Radha Jagadeesan. Automatic abstraction using generalized model checking. In *CAV*, pages 137–150, 2002.

[GL00] Amit Goel and William R. Lee. Formal verification of an IBM CoreConnect™ processor local bus arbiter core. In *DAC '00: Proceedings of the 37th conference on Design Automation*, pages 196–200, New York, 2000. ACM.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[GOR97] Erich Grädel, Martin Otto, and Eric Rosen. Undecidability results on two-variable logics. In *STACS*, pages 249–260, 1997.

[GR06] Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, pages 474–488, 2006.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[GT06] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386, 2006.

[GT07] Sumit Gulwani and Ashish Tiwari. Assertion checking unified. In *The 8th International Conference on Verification, Model Checking and Abstract Interpretation*. Springer, January 2007.

[GW99] Erich Grädel and Igor Walukiewicz. Guarded Fixed Point Logic. In *Proceedings of 14th Annual IEEE Symposium on Logic in Computer Science, Trento*, pages 45–54, 1999.

[HC96] George Hughes and Max Cresswell. *A New Introduction to Modal Logic*. 1996. ISBN 9780415125994.

[HC01] Manuel Hermenegildo and Daniel Cabeza. The PiLLoW web programming library. Technical report, School of Computer Science, Technical University of Madrid, January 2001. http://www.clip.dia.fi.upm.es/Software/pillow/.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *POPL 02 (Principles of Programming Languages)*, pages pp. 58–70. ACM Press, 2002.

[HJS01] Michael Huth, Radha Jagadeesan, and David A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 155–169, London, UK, 2001. Springer-Verlag.

[Hoa69] Tony Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa73] Tony Hoare. Hints on programming language design. Technical Report CS-TR-73-403, Stanford University, 1973.

[HS96] Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 662–681. Springer-Verlag, 1996.

[IEI04] Andrew Ireland, Bill J. Ellis, and Tommy Ingulfsen. Invariant patterns for program reasoning. In *Proceedings of Third Mexican International Conference on Artificial Intelligence (MICAI-04)*, volume 2972 of *LNAI*, pages 190–201. Springer-Verlag, 2004.

[Imm98] Neil Immerman. *Descriptive Complexity*. Springer, 1998. ISBN 0387986006.

[IRR+04] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *CSL'04*, volume 3210 of *LNCS*, pages 160–174. Springer Verlag, 2004.

[JLRS04] Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. A relational approach to interprocedural shape analysis. In *SAS (Static Analysis Symposium)*, pages 246–264, 2004.

[JMG+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, June 2002.

[Kal90] Anne Kaldewaij. *Programming: the derivation of algorithms*. Prentice-Hall, Inc., 1990. ISBN 0-13-204108-1.

[Kle52] Stephen Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952. ISBN 0720421039.

[KLZR05] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Implications of a data structure consistency checking system. In *International conference on Verified Software: Theories, Tools, Experiments (VSTTE, IFIP Working Group 2.3 Conference)*, Zürich, Switzerland, 10–13th October 2005.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[KNR05] Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. An algorithm for deciding BAPA: Boolean algebra with Presburger arithmetic. In *CADE*, pages 260–277, 2005.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[KV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[LAIR+05] Tal Lev-Ami, Neil Immerman, Tom Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE 2005*, volume 3632 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.

[LAMS04] Tal Lev-Ami, Roman Manevich, and Mooly Sagiv. TVLA: A system for generating abstract interpreters. In *IFIP Congress Topical Sessions*, pages 367–376, 2004.

[Lat03] Timo Latvala. Efficient model checking of safety properties. In *Proceedings of the 10th SPIN Workshop on Model Checking of Software (SPIN 2003)*, 2003.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[LF08] Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Compiler Construction*, volume 4959 of *LNCS*, pages 197–212. Springer, 2008.

[LKR05] Patrick Lam, Viktor Kuncak, and Martin Rinard. Hob: A tool for verifying data structure consistency. In *14th International Conference on Compiler Construction*, volume 3443 of *LNCS*, April 2005.

[LQ06] Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *POPL 06 (Principles of Programming Languages)*, pages 115–126, 2006.

[LR07] Patrick Lam and Martin C. Rinard. Static verification of design constraints and software correctness properties in the Hob system. In *IPDPS*, pages 1–6, 2007.

[LRS05] Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. Abstraction refinement via inductive learning. In *CAV*, pages 519–533, 2005.

[Mey92] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0132479257.

[Min61] Marvin Minsky. Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines. *Annals of Mathematics*, 74(3):437–455, 1961.

[Min06] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI*, pages 348–363, 2006.

[MN05] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *CAV 2005*, volume 3576 of *LNCS*. Springer, 2005.

[Mor82] Joseph M. Morris. A general axiom of assignment / Assignment and linked data structures. In *Theoretical Foundations of Programming Methodology*, pages 25–41. D. Reidel Publishing Company, 1982.

[Mor94] Carroll Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, United Kingdom, 1994. ISBN 0-13-123274-6.

[MS01] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01 (Programming Language Design and Implementation)*, pages 221–231, New York, 2001. ACM Press.

[Nel83] Greg Nelson. Verifying reachability invariants of linked structures. In *POPL 83 (Principles of Programming Languages)*, pages 38–47, New York, 1983. ACM Press.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

[NNS01] Flemming Nielson, Hanne R. Nielson, and Mooly Sagiv. Kleene's logic with equality. *Information Processing Letters*, 80:131–137, 2001.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NR03] Karim Nour and Christophe Raffalli. Simple proof of the completeness theorem for second-order classical and intuitionistic logic by reduction to first-order mono-sorted logic. *Theoretical Computer Science*, 308(1-3):227–237, 2003.

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, 2142, 2001.

[PBO07] Matthew Parkinson, Richard Bornat, and Peter O'Hearn. Modular verification of a non-blocking stack. In *POPL 07 (Principles of Programming Languages)*, pages 297–302, 2007.

[PDV01] Corina S. Pasareanu, Matthew B. Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *TACAS*, pages 284–298, 2001.

[PE04] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, Newport Beach, CA, USA, November 2–4, 2004.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS (Foundations of Computer Science)*, pages 46–57, 1977.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RBHC07] Zvonimir Rakamaric, Roberto Bruttomesso, Alan J. Hu, and Alessandro Cimatti. Verifying heap-manipulating programs in an SMT framework. In *ATVA*, pages 237–252, 2007.

[RCK04] Enric Rodríguez-Carbonell and Deepak Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *SAS (Static Analysis Symposium)*, volume 3148 of *Lecture Notes in Computer Science*, pages 280–295. Springer-Verlag, 2004.

[RCK07] Enric Rodriguez-Carbonell and Deepak Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.*, 64(1):54–75, 2007.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95 (Principles of Programming Languages)*, pages 49–61, New York, 1995. ACM.

[RL02] Richard Raimi and James Lear. Silicon debug of a PowerPC™ microprocessor using model checking. *Form. Methods Syst. Des.*, 21(1):79–94, 2002.

[RLS02] Thomas W. Reps, Alexey Loginov, and Mooly Sagiv. Semantic minimization of 3-valued propositional formulae. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, page 40, Washington, DC, USA, 2002. IEEE Computer Society.

[RRL99] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *ESEC / SIGSOFT FSE*, pages 235–252, 1999.

[RSL03] Thomas W. Reps, Shmuel Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *ESOP*, pages 380–398, 2003.

[RSY05] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS (Static Analysis Symposium)*, pages 284–302, 2005.

[SI99] Jamie Stark and Andrew Ireland. Invariant discovery via failed proof attempts. *Lecture Notes in Computer Science*, 1559:271–288, 1999.

[Sis94] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994.

[SK02] Axel Simon and Andy King. Analyzing string buffers in C. In *AMAST (Algebraic Methodology and Software Technology)*, pages 365–379, 2002.

[SR05] Alexandru Sălcianu and Martin C. Rinard. Purity and side-effect analysis for Java programs. Technical report, CSAIL, Massachusetts Institute of Technology, 2005. Amended version of the paper from VMCAI'05.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. In *POPL 99 (Principles of Programming Languages)*, pages 105–118, 1999.

[SSM05] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. of Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 21–47, Paris, France, January 2005. Springer Verlag.

[SW04] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *TACAS*, pages 280–295, 2004.

[vF69] Bas van Fraassen. Presuppositions, supervaluations and free logic. In *The Logical Way of Doing Things*, pages 67–92. Yale University Press, New Haven, 1969.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. SOOT - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[Wre03] Alisdair Wren. Inferring ownership. Master's thesis, Imperial College, London, June 2003. MEng4 Thesis.

[WWA+01] Jongwook Woo, Jehak Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L. Wendelborn. Alias analysis for Java with reference-set representation. In *ICPADS*, pages 459–466, 2001.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[Yor03] Greta Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel Aviv University, March 2003.

[YRS01] Eran Yahav, Thomas W. Reps, and Mooly Sagiv. LTL model checking for systems with unbounded number of dynamically created threads and objects. Technical Report TR-1424, Computer Sciences Department, University of Wisconsin, March 2001.

[YRS+06] Greta Yorsh, Alexander Moshe Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In *FoSSaCS*, pages 94–110, 2006.

[YRSW03] Eran Yahav, Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolution logic. In *Proc. European Symp. on Programming, ESOP 2003*, LNCS, 2003.

[ZKR08] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2008.