An Overview of Program Analysis using Formal Methods

With a Particular Focus on their Relevance for DSLs

Markus Voelter

with

Tamás Szabó and Björn Engelmann

An addendum to the book *DSL Engineering*

Static program analysis refers to determining properties of programs without executing it, relying on a range of formal methods. While these methods have been around for a long time, over the last couple of years, some of these methods started to scale to solve problems of interesting size.
We have used advanced type systems, abstract interpretation, SMT solving and model checking to answer relevant questions about programs written with various DSLs. In this booklet we introduce the methods, illustrate what we have done with them, and describe how we have integrated the analysis method and existing tools with languages and IDEs.

Note that this booklet documents the authors' experience. This is not a scientific paper. There is no contribution. The aim is to explain and illustrate.

Version 1.0 | July 18, 2017

Contents

1. Front Matter

For many practitioners, even though they might not clearly know what the term formal methods means, it is "useless computer science theory". Some of the authors of this booklet have held this opinion as well, not too long ago. However, this"computer science theory" is becoming more and more relevant to practitioners, especially to those working with DSLs, for a number of reasons:

Most importantly, *customers ask for it*. We are starting to use language engineering in domains where safety and reliability is important. Example include medicine and automotive. These domains would like to benefit from ways of "proving" that something works. They might not always be willing to pay for it, but the interest is there, and we have sold these approaches successfully.

Second, the *analysis tools are becoming better*. This means that they can handle more complex cases and scale to larger problems. For example, the Z3 solver can work with millions of free variables and still find solutions in seconds. There are still limits to both, and understanding those can be hard, both for the user and the language developer.

Finally, there is a strong *synergy with models and language engineering*. One major reason to use models in the first place is that they can express domain semantics directly, so no "semantic recovery" from low level abstractions is required. Language engineering allows us to tailor (extend, restrict, adapt) languages in a way that makes those aspects of the semantics that are required by a particular analysis first class. This reduces the overall complexity of the analysis, making the analysis easier to implement (for the analysis developer) and more accessible to the end user.

Target Audience and Prerequisites When dealing with formal analyses in the context of languages, we distinguish two roles. The *end user* is a developer who uses a (domain-specific) language and the formal analyses it comes with to build reliable and safe systems. He treats the analyses as a black box to the degree possible – just another check provided by the IDE. The *language developer* creates the languages used by the end user, and also develops the analyses. He understands language structure and semantics, and understands the algorithms involved in the analyses (or how to use existing analysis tools) and integrates those with the language and the IDE in a way that makes their use as simple as possible for the end user. This booklet primarily targets the language developer by explainig the basics of a number of formal methods and how, in principle, they can be integrated with languages and IDEs. However, the end user can also benefit from reading (at least parts of) the booklet to understand what these methods can do to reduce the number of errors in models or programs.

In any case we assume a basic functionality with the implementation of languages. Terms such as abstract syntax tree, IDE, type check or interpreter are assumed to be known and are not introduced.

Style and Depth This booklet is meant to provide a high-level overview over some of the available formal methods, and what they can be used for. After reading the booklet, readers will be able to judge whether a particular formal method can be interesting in their domain, and they will have a rough understanding how they can be integrated with DSLs. This booklet is very pragmatic and example-driven. It does not discuss much of the theoretical background of each method – their are books full of details about each of them, for example [32] and [30] – but hopes to contribute to a rough understanding of the basic idea. This is not a scientific booklet, there is no "contribution". Instead, the booklet aims to explain and illustrate.

Context This booklet essentially summarizes the authors' (and their colleagues) experiences with implementing analyses and verifications in the context of DSLs. The vast majority of practical experience – and thus, the examples – are based on work with Jetbrains MPS[1]. MPS is a language workbench, i.e., a tool for efficiently developing domain-specific and general purpose languages and their IDEs. A team at itemis has spend dozens of person years building a wide variety of languages based on this tool. The major case study was mbeddr [47, 48], a set of domain-specific C extensions optimized for embedded software development. All the publications can be found on the mbeddr website[2]; the condensed experiences of 10 person years of mbeddr development in terms of language engineering can be found in this paper [49]. Several examples are also based on KernelF, a functional language intended for embedding into DSLs developed recently. A preliminary language documentation can be found online[3].

Languages vs. Data Structures The authors of this booklet consider themselves language engineers, meaning that they develop (domain-specific) languages and IDEs. They look at the analyses discussed in this booklet as a means of making programs written with these languages "more correct". So for readers who are language engineers as well, the concerns and examples in this booklet should sound familiar. However, as Fig. 1 shows, at the core of a language we can find data structures (meta model) and their semantics. And in fact, the (concrete) syntax aspect of languages is completely irrelevant for the analyses discussed in this booklet, and we cover IDE concerns only superficially. Essentially we look at languges as data

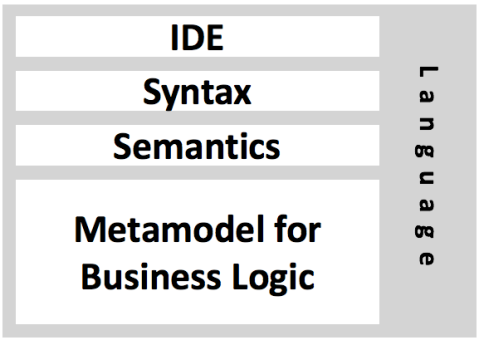

Figure 1. Language == Meta model/Structure, Semantics, Syntax and IDE.

structures that are instances of a well-defined schema, which means that all readers who work with structured data in the broadest sense, and who want this data to exhibit certain properties, should benefit from reading this booklet.

The Meta Level This booklet looks at the use of formal methods to verify *programs* expressed in a particular language, written by end users. Obviously, since language definitions are also just programs written in particular language, formal methods can also be used as part of the definition of languages. Examples include proving the properties of model transformations [25] or the semantics of languages [44]. However, this is not the scope of this booklet and we consciously avoid talking about this aspect.

Structure of the booklet Sec. 2 introduces general ideas about program analyses and verification, including the difference between execution and analysis, verification vs. testing, various criteria by which verification approaches can be classified and compared, and considerations when integrating a verification tool with a language implementation.

The rest of this booklet is structured into four chapters that each introduce a practically relevant formal verification approach[4]: type checking in Sec. 3, abstract interpretation and data flow analysis in Sec. 4, SMT solving in Sec. 5 and finally, model checking in Sec. 6. Each section discusses the following aspects, even though each section has its own structure in terms of subsections.

What is it? What are the conceptual or mathematical basics? What are the conceptual limitations?

What can it be used for? In the context of DSLs and program verification, what can the approach be used for?

Real-world use. In real-world systems, what has been done with this approach? Does it scale?

[1] `https://www.jetbrains.com/mps/`

[2] `http://mbeddr.com`

[3] `http://voelter.de/data/pub//kernelf-reference.pdf`

[4] Additional formal methods exist, of course, but as a consequence of our lack of experience, we do not discuss them in this booklet.

2. Introduction

2.1 Terminology

Formal Methods The term "formal methods" is not well-defined. We have heard people use it to denote techniques that rigorously formalize mathematical models of their object of study and use mathematical proofs. Others call the combination of a symbolic program verification methodology (e.g., symbolic execution, weakest precondition calculus or Hoare Logic) with an interactive theorem proving backend a "formal method". This combination is optimal in terms of soundness and completeness, these tools are only semi-automatic: they require manual effort as well as extensive training in the used formalisms and mathematical proof, as the user has to manually construct proofs about his programs. When interpreted loosely, the term encompasses most of theoretical computer science, to the degree it is used for verifying the correctness of programs.

Program Verification and Analysis Many people (including us) use the term "formal methods" as a synonym for "program verification", which is that part of theoretical computer science that deals with deriving formal guarantees about a program's behaviour. Static verification is that part of program verification that deals with deciding program properties for all possible executions of the program (i.e., not by running the program). This is in contrast to runtime verification, which monitors properties of a particular execution at runtime.

Program analysis refers to any analysis of a program, including those that do not contribute to correctness. For example, clone detection, timing analysis or style checkers can be seen as program analyses, even though they are not program verifications in the sense that they help with program correctness. These analyses are out of scope of this paper. Thus:

In this booklet, we introduce practically useful program analysis and verification techniques and illustrate their use in the context of DSLs.

In the remainder of this booklet, when we use the term program analysis, we refer to analyses that help with correctness. We use program verification program analysis interchangeably.

Use of Verification Results Once results have been obtained, they can be used in various ways: they can be presented to the user, for example, through source code annotations in an IDE, encouraging the user to change the program (presumably in order to make it "more correct"). For example, the user might get a warning that some part of the code is unreachable or an error[5] that two guards in a switch-case-statement overlap. In the former case the user might want to delete the dead code, and in the latter case, the user must change one of the guard expressions to remove the overlap. Alternatively, analysis results can also be used by downstream tools directly. For example, one analysis might depend on the result of another one, or a compiler/generator might use the unreachable code information to not generate any binary code for the dead source (effectively removing the source code implicitly).

[5] In this booklet we gloss over the differences between defect, fault, failure and bug; see [51].

2.2 Execution vs. Analysis

While in the last section we were also discussing about things requiring a theorem prover (overlap in switch-case statements), this section and the next are clearly about program analysis. I would hence suggest putting them into an appropriately named section.

For the purpose of this discussion, we consider a program to be an implementation of an algorithm that takes input values, performs a computation, and then produces output values. The program abstracts over these values, representing them as (usually typed) variables. Note that programs are hierarchical; for this discussion, we consider a program to be a function.

As developers, we are used to executing programs. Execution means that we supply a particular value for each input variable, and then run the algorithm encoded in the program. The parts of the algorithm that are actually executed may depend on the values (a consequence of conditionals in the code). As a result of execution, we get particular values for output variables.

Program analysis, in contrast, attempts to derive properties of a program for all possible input values, in order to characterize the set of all possible output values or to find problems in its internal structure. A brute force way of analysing a program is to actually execute it for all values, one at a time, and then generalize from the collected output values or other data collected during the executions. While this is possible in principle, it only works for trivial programs as the set of inputs may be infinite in general and – even when finite – is often prohibitively large in practice. Program analysis attempts to derive the same results through a more "clever" approach.

In contrast, in runtime analysis (or runtime verification), the analysis is performed at runtime. For example, an instrumented program could collect execution traces, memory allocations, or the execution frequencies of statements, for example, to find out which parts of the program are worth optimizing. Another example would be a state machine, in which, when two transitions are ready to fire at the same time (because their guards overlap), a runtime error is thrown. In both cases, data is collected (or errors are found) only for particular execution of the program, hopefully during (systematic)

testing. This is in contrast to the static analysis discussed in this booklet: a static program analysis can prove the absence of particular errors (for all possible executions of the program), whereas a dynamic program analysis can show that some errors actually occur. By this definition, runtime analysis or verification is a misnomer; it should instead be called runtime monitoring or runtime diagnostics.

Value Analysis A value analysis derives properties of values computed in a program. Examples include type checking (what kind of values are possible?), interval analysis (What are the minimum and maximum values?), `null` analysis (can this value ever be `null`? Do I need a check to prevent `NullPointerException`s?), or constant propagation (is a condition always true or false? Can the conditional hence be omitted?). Many aspects are principially undicidable statically, i.e., when they are based on input values that are not know before runtime. Thus, such analyses often rely on heuristics and/or program annotations provided by the user[6].

Location Analysis Location analysis aims to derive properties of program locations and/or program parts. Examples include reachability (can this part be executed *at all*?); the dead code analysis mentioned earlier is an example of this), coverage (for a given set of inputs (tests), are specific parts of the program executed?), timing (how long does it take to execute this part of the program) or resource consumption (how much memory does this function need?). Note that a location analysis might require value information as well. For example, whether a part of a program is executed may depend on whether a condition can ever become true. Also, the worst case execution time of a while loop obviously depends on the (initial) values of the variables referenced in its condition.

This booklet focuses on analysis (as opposed to execution) and mostly on value-based analysis.

2.3 Overview over different Approaches

An important part of integrating an analysis into a DSL is choosing the right formal method for the job. As a basis for this decision, we will in this section provide an overview of the most relevant formal methods. For the purposes of this tutorial, we identify the following four major approaches. We provide a quick overview here, and then elaborate in the rest of this booklet. In the book Principles of Program Analysis by Nielson et al. [32] all four are explained in detail.

Type Systems Type systems are well known to programmers; they associate a type with a program node. That type is used to check program correctness beyond the structures defined by the AST[7]. For example, while a plus operator structurally works with two arbitrarily structured expressions `e1 + e2`, from a type system perspective, `e1` and `e2` must either both be of type `number` or of type `string` in a typical language. Typical tasks of a type system are calculation of simple types (the type of `2` is `number`, deriving types from other types (a variable reference's type is the type of the referenced variable), computing the supertype of a set of types (the supertype of `int` and `real` is `real`) and checking expressions for type compatibility (the plus example from before). In the vast majority of cases, the type of a program node can be computed by inspecting a single program node (e.g., number literals) or by looking at simple structural relations between program nodes (e.g., function call, or the two children of plus). Type systems are discussed in Sec. 3.

Note that often, name analysis, i.e., the challenge of binding names mentioned in the program to its definition, is considered part of the type system as well. However, we do not consider name analysis in this booklet, mainly because our experience is centered around languages built on MPS where name analysis is not an issue.

Abstract Interpretation In contrast to type systems, which rely mostly on the structure of a program to compute and check types, analyses based on abstract interpretation rely on (variants of the) execution semantics. For example, a data flow analysis that checks for uninitialized reads will "run"[8] the program and see whether any variable may be read before it is ever written; or a constant propagation analysis "runs" the program to see if conditions in an `if` statement can be statically determined to always be true or false. We have put the word run into quotation marks because, to make analyses based on abstract interpretation efficient, programs may be run partially (local analyses in a single function), with approximating data types (lattices) or with simplified operational semantics (ignoring exceptions). We discuss abstract interpretation and data flow analyses in Sec. 4.

Constraint Solving Constraints are similar to functions in that they relate various values and variables with each other. A function `f(x) = 2 * x` computes a value `f` when given a value `x`. Functions are executed "from right to left": when provided values for all arguments, the resulting value(s) is calculated. Functions can call other functions, but they are still executed in a linear, nested, and perhaps recursive way.

[6] Or information that can be derived from domain-specific abstractions in the program – this is the synergy with language engineering again.

[7] The abstract syntax tree (AST) is a tree or graph representation of the structure of a program. Notational details such as keywords or whitespace are elided. We provide more detail on the AST and other derived data structures in Sec. 4 and Fig. 5.

[8] Technially: it calculates along the control flow paths.

Constraints express relationships between variables, for example, `2 * x == 4 * y`. A constraint solver computes values for all involved variables (here: `x` and `y`) so that the constraint is true, such as `x=2` and `y=1`. The solver can do this even when multiple constraint equations and many variables are involved. Importantly, in contrast to functions, constraints do not imply a "direction". This means that solvers do not just "execute" programs, they search/find solutions to questions that make a program become valid (often called forward evaluation, or just solving).

We are primarily concerned with SMT solvers in this booklet. They can do this for complex Boolean expressions, as well as a wide range of other theories, such as numbers or lists. We discuss SMT solving in Sec. 5.

Model Checking For formalisms that have side-effects, such as state machines or non-functional programming languages, constraints have to consider the evolution/change of state over time; the constraint language, as well as the tools for checking the constraints must be able to handle notion of (logical, discrete) time. Model checking is such a formalism. For example, in a functional language, one might express a postcondition of a function as `post: res >= arg1 + arg2`. In a language with side effects, one might have to express that a function increments a global variable, and to do this, one has to refer to the old value of that variable: `post: global = old(global) + 1`. The `old(..)` notation refers to the value of the `global` variable *before* the execution of the function – essentially, it looks back in time. A second example are constraints such as "After the state machine has been in state A, it will always eventually reach state B". This constrains the sequence of states that are valid for a given state machine. Such constraints can be checked using model checking; model checkers work on transition systems (aka state machines) and verify constraints expressed in temporal logic formulas. The one mentioned above can be expressed as `AG((state == A) => AF(state == B))`, where `AG` and `AF` are operators from temporal logic that quantify over time and execution paths. We discuss model checking in Sec. 6.

2.4 Testing vs. Verification

Testing refers to *executing* a (part of a) program for specific input values and asserting that the behaviour (i.e., often the outputs) corresponds to some expectation[9]. Verification, in contrast, relies on analyses, i.e., the programs's behavior for all possible executions/inputs (as we have discussed previously). In other words, testing shows the presence of bugs whereas verification proves their absence.

As a consequence, testing suffers from the coverage problem, which means that we can only be sure that the program behaves correctly for those inputs that are covered by the tests. There are many means of measuring coverage [52] including coverage of input vectors, coverage of all paths/lines in the code, or coverage of all decisions. The software engineering community disagrees over which coverage measure is appropriate. What is an appropriate minimum coverage value for us to be able to trust the code (i.e., be sure that it is correct) depends on the risk associated with faults – always testing everything to 100% coverage is too expensive and hence not feasible.

This sounds like verification is *always* better than testing. However, as we will see in this booklet, verification has limitations regarding computational complexity (things get slow), and complexity regarding the users skills. In particular, writing correct specifications (that specify a program's behavior for all possible cases) is much more challenging for practitioners than writing test cases, which means that is likely that a specification is wrong or incomplete – thus, voiding the theoretical benefits of verification. Reasons for this include the fact that often, a separate language must be used for specifications, that it is usually more complicated to think about *classes* of behaviors rather than particular instances, and that practitioners just aren't used to it.

Another difference between testing and verification is the scope: verification ensures that the program is correct according to its specification under some assumptions, e.g. that the hardware is correct. Testing has no assumptions, but the limitation that only one specific execution is considered.

It is obvious that verification do not replace testing, but rather complements it. A middle-ground is test case generation: using an analysis, one determines input vectors that satisfy one of the relevant coverage criteria, as well as possible preconditions for the test subject. We then actually run the tests for those vectors. We discuss this briefly in Sec. 5.6.

2.5 Correct-by-Construction

If it is impossible to write a defective program, then there is no need to verify or test that a program is correct. A language/system that allows only the implementation of correct programs is called correct-by-construction. While this is an appealing idea, it has limitations.

Let us consider the case of implementing state-based behavior. In C, state-based behaviour might be implemented as a state machine simulated with a `switch` statement. One particular error is to forget the `break` statement in the `case`s, leading to unwanted fall through.

[9] We are consciously avoiding the term "specification" here, because usually there are no formal specifications for programs available if testing is the primary method of ensuring the correctness of a program.

This problem could be found by analysing the code (ensuring that every path through the code in the `case`s either ends with a `break` or a `return`). Alternatively one can use a C extension that provides language concepts for encoding state machines directly, such as those provided by mbeddr (mbeddr is a set of language extensions for C to facilitate embedded software development [47, 48]). Because the abstractions encoded in the additional language constructs are aligned with the domain (state-based behavior), the programmer does not have to insert the `break`s manually. So, *regarding this particular error*, the use of the C state-machines extension indeed guarantees correctness-by-construction.

However, the programmer can still implement a state machine that has missing transitions or a "loop" because the machine executes ϵ-transitions[10] into the same state over and over again. The state machine language will very likely not *prevent* the user from these mistakes; thus it is *not* correct-by-construction for these errors. Similarly, consider the infamous C `strcpy` function that copies a string from one buffer to another. `strcpy` continues copying characters until it finds a `null` character in the source buffer; if none is found, it copies forever and overwrites memory beyond the (limited size) target buffer, leading to security vulnerabilities. Removing `strcpy` and forcing users to use `strncopy`, where the length of the buffer is given by an additional parameter, is a good idea; however, users can still pass in the *wrong* length, also leading to an error. So, while the forced use of `strncpy` avoids some errors, it does not prevent all.

In both cases – state machine and `strcpy` – we can combine correctness-by-construction with an analysis: we can implement an analysis on the state machine abstractions that finds ϵ-transitions into the same state and reports them as an error. Similarly, we might want to analyze C programs to see if the length argument to `strncpy` is correct. It should be obvious that the former analysis is much easier to build than the latter. This suggests that, even though adding domain-relevant concepts (such as state machines) as first-class language constructs does not lead to correctness-by-construction for all possible errors, it likely makes (relevant) analyses simpler. We discuss this synergy between analysis and language engineering in Sec. 2.10.

In practice, the distinction between correctness-by-construction and analysis/verification is blurry. Consider the example above where an IDE-integrated analysis reports ϵ-transitions into the same state essentially in realtime. The user can technically write buggy code (in the sense that the language does not actually *prevent* him from expressing the error), but because he gets the error immediately (i.e., without invoking a special tool or maybe adding additional specifications to the code), this is just as good as a formalism that syntactically prevents the user from making the error. Thus, we consider IDE-integrated verification techniques to contribute to correctness-by-construction.

There is another problem with correctness-by-construction. Very often, correctness-by-construction is achieved through better (higher) abstractions. To execute them, these typically have to be translated to a lower-level implementation. The problem is that this transformation might be buggy[11], i.e., the generated code might not have the same characteristics as the more abstract model that guaranteed the absence of some particular (class of) errors. For example, the generator that transforms the C state machine extensions into low-level C (e.g., `switch` statements), might "forget" to generate the necessary `break` statements. While test case generation can help (the test are derived from the model but executed on the generated code, thereby "propagating" the checks down), ensuring transformation correctness is a challenge that is beyond the scope of this booklet.

2.6 Derivation vs. Checking vs. Synthesis

Formal methods can be used several ways; which of those are supported depends on the method itself – in particular, synthesis is not supported by all methods.

Derivation The most straightforward one is deriving a property. For example, for an expression `2 + 3.33`, the type of `+` is derived to be the common supertype of `2` and `3.33`, which will be `float` (the type of the `3.33`). This is the classic case of program analysis: it computes a property of an expression, declaration or statement and "attaches" it to the corresponding AST node. It can then be shown in an IDE as a error or warning ("this condition is always true") or it can be exploited in a transformation (a condition that is always true can simply be omitted).

Checking Checking compares two properties, one of them typically specified by the user and the other one automatically derived. For example, a variable declaration could be specified as `int x = 2 + 3.33;` The type of the expression is derived, and then checked against the specification `int`. In this sense, checking is very much related to derivation, because a check typically first involves the derivation. The check itself is usually relatively simple.

Synthesis Synthesis refers to *finding a program* that satisfies a specification (the reliance on a specification is a commonality with checking). To continue with

[10] "Automatic" transitions that are triggered without an incoming event.

[11] This is true for any compiler; but it is especially critical if the source language suggests that programs are provably correct, because users might not do a lot of testing because of this guarantee

the example above, one could write a program that contains a variable declaration `int x = ??;` where the `??` mark represents a hole in the program that needs to be completed. The task of the formal method is to synthesize a replacement (or completion) for the hole that satisfies the specification. There are two challenges with this. First, the method must be able to actually find solutions; whether this is possible, and how fast, depends on the math that underlies the formalism and the sophisticatedness of the tool. Second, one has to define a sufficiently narrow specification, because the synthesis usually finds the simplest solution (`int x = 0` in the example); this solution, while correct, might not be useful. A narrower specification has to be written, and the synthesis repeated. For example, you could write `int x = ?? + ??;`, which might force the synthesizer to fill the two arguments of the plus operator.

2.7 Specifications

A specification describes *what* something does, but not *how*. Another way of expressing this is that a specification defines a (black-box) contract but presupposes nothing about the implementation. A specification achieves this by precisely describing a property of an artifact (in our case, a piece of code), while not saying anything about other properties. Of course, the boundary between what/contract and how/implementation is arbitrary to some degree: for example, a specification might specify that an array should be sorted, leaving the actual algorithm to the implementation; but a specification might also prescribe the algorithm (e.g., quicksort), but not define how it is implemented in a particular programming language.

Good Specifications In some sense, a specification is redundant to the system it specifies: it specifies behavior which the code already implements. A verification can be built so as to check if the implementation fulfils the specification, and if the two do not represent the same (or compatible) behaviors, the verification fails. The reason for the failure can either be an error in the implementation *or* in the specification. Often one assumes that the implementation is wrong because a good specification is simpler than the implementation (in terms of size or accidental complexity, for example, because it ignores non-functional requirements such as performance), but nonetheless expresses the relevant property. Because it is simpler, it is easier to relate to the original (textual or perceived) requirements, and hence it is less likely to get the specification wrong than the implementation. Let us look at examples:

```
fun divBy2(x: int) post res = x * 2 {
  x * 2
}
```

In this case, the postcondition (i.e., specification of what the function returns) is identical to the implementation of the function. Unless the specification is taken from some interface defined by somebody else, it does not add value[12]. However, consider the next example:

```
fun sortAsc(l: list<int>): list<int>
  post l.size == res.size
  post forall e in l { e in res }
  post forall i, j: int[0..l.size]
              { j > i => res[j] >= res[i] }
{
  .. 20 lines of intricate quicksort ...
}
```

Here, the specification concisely expresses that (1) the resulting list `res` has the same size as the input list `l` (2) contains the same elements and (3) must be sorted in an ascending order. This is much more concise and understandable than 20 or so lines of Quicksort [18] implementation. It is also generic in the sense that it works for many different implementations, which is another characteristic of a good specification.

You probably think that this specification is current and complete. But in fact, it is not: it does not say anything about the number of occurrences and does also not exclude duplicates. e.g., `1, 1, 1, 2` might be sorted to `1, 2, 2, 2` which would fulfil the specification. Another constraint is necessary. This illustrates nicely that writing correct and complete specifications is not a simple task!

Ideally, a specification is declarative, so it itself can be verified for internal consistency[13]. For example, if the postcondition would have been written as

```
post forall i, j: int[0..l.size]
            { j == i => res[j] > res[i] }
```

then this could be identified as inconsistent, because, since `res[i]` and `res[j]` refer to the same value, one cannot be greater than the other one.

Exploiting Redundancy Some specifications cannot be used as the input to a static program analyzer that proves properties for all possible executions of a program (because of the use of a formalism that is not supported by the verifier, because it is too complex or because it is not complete). In this case, we can at least exploit the inherent redundancy. For example, in order to ensure the semantics of a language, we can implement the language with a (simple) interpreter and with a (more complex, fast) code generator. We consider the interpreter the

[12] Another reason why the specification might be useful is when the specification language has different semantics. For example, the implementation, if written in C, might overflow, whereas the specification might not.

[13] This relates to theorem proving which relies on a sequence of proofs, which in turn rely on axioms or previously proven rules. Theorem proving is another formal method that can be used to prove programs correct, but it is beyond the scope of this booklet. See [46] and [33] for an introduction.

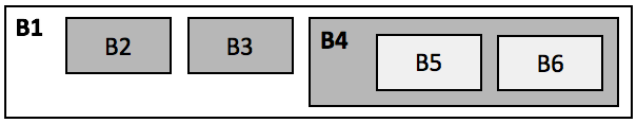

Figure 2. An example of a hierarchical system.

"specification". We can run all programs with both, and if one fails, we know that there is either a bug in the interpreter or the code generator. Using the simplicity argument from above, we assume the code generator is defective and we can try to fix it. Note that, similar to testing, this approach is not general: we can only prove the absence of bugs for those programs that are actually written and executed in both environments.

Decomposition Specifications themselves might become big and/or complicated. This leads to two problems. First, the assumption that it is "easy to get right" might be called into question. Second, the verification tool might run out of steam when trying to verify it. Solving this problem relies on a trusted software engineering approach: decomposition. Consider Fig. 2.

To verify this system, one starts by verifying `B4` using the (assumed to be correct) specifications of the internals (`B5, B6`). Once `B2, B3` and `B4` are verified this way, we can verify `B1`, again, assuming the internals are correct. For each level, we only have to consider the *direct next level*, but we can ignore the internals of that next level. This limits overall verification complexity, both for the tool, and for the user.

Synthesis vs. Code Generation Now that we understand that good specifications are ideally simple and declarative, the challenge of program synthesis becomes obvious: in the example above, you would write:

```
fun sortAsc(l: list<int>): list<int>
  post l.size == res.size
  post forall e in l | e in res
  post i, j: int[0..l.size] | j > i => res[j] >= res[i]
{
  ??
}
```

The task of the the synthesizer is to "fill the hole", automatically coming up with an implementation that performs sorting of a list, ideally in an efficient way. And this would have to work for any specification you can express with the specification language!

This is different from code generation in the way we use it for DSLs. In that case we would perhaps write:

```
fun sortAsc(l: list<int>): list<int> {
  sort l, ASC;
}
```

Here, `sort` is a language construct, so the code generator understands the *specific semantics* of the concept. The generator contains predefined mappings from the language construct to to-be-generated lower level code, implementing the code for the `sort` construct trivially. In contrast to program synthesis, the generator would have to contain specific, manually written generator logic for each of these predefined constructs. So synthesis has to add additional information to "invent" an algorithm whereas generation has the algorithms already add hand, at least in the generator. This means that building generators for specific language constructs is easy (and practitioners do it all the time), and program synthesis is hard (and is done only in very narrow circumstances). In this booklet, most of our use cases for formal methods center around derivation and checking, and only very limited forms of synthesis.

2.8 Evaluation Criteria

No single formal method excels in every respect (and due to some hard theoretical limitations, it is highly unlikely that there will ever be one that does). We consider the following criteria to be relevant when choosing from or evaluating various formal methods.

Automation The degree to which a formal method can be automated is of great relevance for its industrial application. Due to its paramount importance, we discuss in more detail in Sec. 2.9.

Soundness Soundness means that a formal method provides a strong correctness guarantee unless it flags an error. When applying a sound formal method to a program and not finding any errors, one can safely consider the program to be correct (relative to the specification, which might or might not be correct itself). In some cases (e.g., bounded model checking), the soundness guarantee depends on configuration parameters (see Sec. 2.13) – the higher the bound, the stronger the guarantee provided. If chosen wrong, the method might seem sound, but isn't in fact. An alternative definition of soundness is the absence of false negatives. A sound formal method hence cannot "overlook" or otherwise fail to detect errors present in the program.

Precision We call the probability that a formal method is able to establish a property for a given program it's precision. For example, due to the approximate nature of their analyses, type systems and data-flow analyses are often not able to prove properties even though they actually hold for the given program. The higher the precision of a method, the fewer false positives it produces. The extreme of precision is completeness, which is the complete absence of false positives and hence the counterpart of soundness.

Applicability Applicability describes the degree to which a formal method is applicable to different programming paradigms. Often, there are restrictions regarding certain language features. For example, dependent types [50] and refinement types cannot deal with

side-effects and are hence only applicable to functional languages, so this advanced form of type checking cannot be used for imperative languages.

Flexibility The degree to which a formal method can be used to verify different properties. Some methods come with a predefined set of properties, whereas other methods lets the user define a wide range, or an arbitrary set of properties.

Applicability and flexibility are sometimes hard to distinguish: applicability refers to the formalisms/-paradigms *to which* an analysis can be applied. flexibility, in contrast, describes what can be analyzed on programs to which the method is applicable.

Performance/Scalability How much time (or instructions) is required for an analysis and how does this depend on the program size or the property's complexity? Many formal methods suffer from problems. For some analyses, the problem can be adressed by manually decomposing the structure of the program and the specification in a way that allows modular verification. For other analyses this is not (easily) possible.

Unfortunately, any particular formal method involves tradeoffs between those criteria (for example between precision and scalability or between flexibility and automation). We will discuss the criteria and the involved tradeoffs for each formal method in their respective sections.

2.9 Degree of automation

Analyses can be grouped with regards to the degree of automation from the perspective of the end users. We emphasize "end user", because, from the perspective of the language developer, a lot of manual integration work may be required; we discuss this in Sec. 2.11.

Specification-based analyses perform the verification of a property automatically, but require the user to express the property that should be checked. The property is specified in a language that is suitable for the particular analysis. The simplest example of this approach is type checking: a user specfies the (expected) type of a variable and the type checker verifies that a value (potentially a complex expression) is compatible with this expected type. Another example is model checking, where a user might specify temporal properties ("after the state machine was in state `A`, it will always eventually reach state `B`"), and the model checker tries to prove that the property is true.

One challenge in this approach is that it is additional effort for the user to write these properties. While a good design (and integration) of the property specification language helps, it is still often a challenge to make users write the properties. For example, the pre and postconditions in mbeddr components were used rarely, even by experienced developers [48], even though mbeddr supports fully-integrated verification of these contracts through CBMC [21].

Another problem with specification-based analysis is that the verifier will only address those properties that are actually specified by the programmer[14]. It is almost always manual work to derive these formal properties from requirements (if clear requirements exist in the first place). This means that specification-based verification has a coverage problem.[15]

Fully automated analyses report errors or warnings on programs written by a user without any additional (analysis-specific) activities performed by the user. For example, for a state machine, a model checker can verify that all states are potentially reachable (dead states are always an error), or an SMT solver can check for a set of Boolean expressions that no two of them are identical.

Fully automated analyses can also be seen as specification-based analyses where the properties are implicit in the language construct that is being verified (see next subsection for more details on this idea). Again, consider model checking: it is an implicit property of a state machine that all states must be always eventually reachable, i.e., the state machine has no dead states. Note however that fully automated analyses might still require configuration or tuning parameters for the analysis backend (see Sec. 2.13).

Interactive Analyses are those where the tool and the end user conduct an interactive, often iterative, back-and-forth session. For example, a user might specify a to-be-proven theorem, the tool automatically simplifies the theorem by applying simplifications based on other theorems, but then requires additional specifications or decisions from the user before it can continue with the proof. Interactive verifications are outside the scope of this booklet.

In this booklet, we focus on fully automated as well as specification-based analyses.

2.10 Synergies with Language Engineering

Above we have described the need for specifying properties (and getting them correct!) as one of the major problems for the adoption of static analyses. On the other hand, we have also pointed out that analyses that do not require explicit specification of properties because the properties are implicit in the language constructs used to write programs are easier to get used in practice.

[14] Some verifiers, such as KeY [3], automatically flag every non-specified behavior as an error.

[15] The coverage program can be seen as smaller because a property is a "test case for all possible executions" and hence more powerful. On the other hand, full coverage of a specification cannot be proven – test coverage can at least be measured.

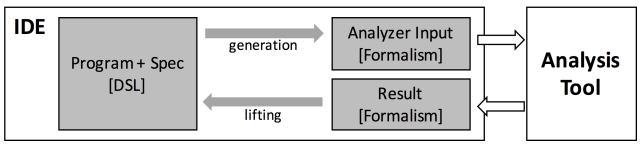

Figure 3. Typical architecture of integrating verification tools.

So, if we assume that the subject language can be changed or extended (as is the case in the context of language engineering and language workbenches), this leads to an obvious conclusion: adapt or extend the subject language with constructs, that imply analysis properties and/or let the user express the properties in a user-friendly and integrated way.

An example is a decision table, such as the one shown in Fig. 12. The decision table language construct implies that both the conditions in the row headers and the conditions in the column headers must each be complete (any combination of value must be matched by the table) and free of overlap (any set of inputs must match at most one condition). The lower-level representation of decision tables (a bunch of nested `if` statements) does not imply this. However, a verifier integrated into the language can perform the corresponding verifications for every decision table it encounters without any *additional* specification. We revisit this particular example in Sec. 5.

2.11 Analysis Architecture

Analysis tools are typically quite sophisticated: they embody years or decades of development effort to perform particular analyses reliably and fast. This means that in most cases, one will want to integrate existing verification tools instead of developing your own verifications from scratch. In addition, each verification formalism expects the problem to expressed in a specific way (e.g., model checkers expect the to-be-verified system as a transition system, aka. state machine); and in addition, each tool has its own specific syntax (all model checkers expect a transition system as the input, but each tool has a different syntax to express that transition system). This leads to the tool integration architecture shown in Fig. 3.

The program is expressed in whatever DSL has been chosen for the domain. The specification, if necessary, is also expressed in a suitable DSL, either separate (but relating to) the program, or syntactically integrated. A transformation maps (the relevant parts of) the DSL program to the input of the verifier and forwards it to the external tool. The verifier performs the analysis and reports a result in terms of the verifier's input formalism and in some tool-specific syntax. This then has to be lifted back to the DSL for it to make sense in the context of the original program. Note that for reasons of reuse and/or complexity, the transformation might be performed in multiple steps.

Note that the transformation and lifting encompasses a semantic/algorithmic aspect and a technical one. In almost all cases, the program as expressed in the DSL has to be represented in the tool's formalism (e.g., a procedural program has to be represented as a transition system for model checking). Similarly, the results have to be lifted back. This is the semantic aspect. The technical aspect then involves, for example, generating an actual text file and feeding it to an external process, or calling an API of an in-process library.

The architecture shown above has a couple of challenges that have to be implemented by the developer of the language and its integrated verifications.

Maintenance of Derived Structures Whenever the program changes, the derived structures have to be updated. For small, local analyses, the derived structures can be rebuilt completely, for every program change. However, for larger programs, these derived structures must be maintained incrementally. Currently, we do not have a means for such incremental transformations ("shadow models"). If an external tool is used, an additional problem is that the tool typically does not provide an incremental API and requires a completely new model for every verification run[16]; so even if we could maintain the tool's input model incrementally, the verification would still happen from scratch each time.

Lifting of Results Analysis tools report results on the level of the tool's abstraction. For example, an SMT solver will report results on the level of the constraints supplied to the solver. However, to be meaningful to the user, the results have to be lifted back to and expressed in terms of the abstractions of the DSL: the gap that is crossed by the creation of the tool-specific structures has to be bridged in the reverse direction. While this can be non-trivial, it can be simplified by suitable intermediate abstractions for which, when they fail, the meaning in terms of the DSL is clear (an example is the solver intermediate language introduce in Sec. 5.2), by clever naming (so that, based on an analysis result, one can link back to the names in the program) or by embedding information in the lower level representation (similar to an asynchronous completion token [7]). If the down-translation is done in multiple steps, lifting back the results can also be simplified.

In some cases it might also be feasible to perform the analysis directly on the code, with no translation of the program into another formalism (although, very likely,

[16] A version of CBMC has been built that, before verification of C sources, first diffs the new input with the previous one and then tries to re-verify only the changed parts. Boogie [22] also supports incremental verification.

the analysis tool itself then performs such a translation). An example of this approach is model checking on C level using tools like CBMC.

2.12 Level of Confidence

One usually applies formal methods in order to gain strong correctness guarantees for subject programs. When designing a verification system for a language, it is hence important to understand what level of confidence is expected from it, since this has direct implications on the methods, components and tools that can be used in its construction and operation (cf. the previous subsection).

Tools When deciding on tools, apply the standard metrics from software engineering to judge its maturity: (1) How long has a tool been available? (2) How many people are working on it? (3) Is it still being maintained actively? (4) How many bugs have been found / fixed in it? (5) How large is a tool's user base?

Method When developing a verification method, it is important to realize that the guarantee provided by the system as a whole can only be as strong as the weakest link in the process: for example, when using an existing SMT solver with a strong soundness guarantee, one should place special emphasis on developing the translation from the DSL to the solver's input language rigorously to avoid introducing unsoundness. When developing a program analysis based on abstract interpretation, it is important to prove one's abstraction sound with respect to the semantics of the programming language because otherwise the analysis will not provide a meaningful guarantee. When developing an interpreter for a DSL, it is important to ensure that it properly implements the DSL's semantics because otherwise the analyses one wishes to perform might have a slightly different understanding of the language's semantics and thus fail to provide relevant guarantees.

Bootstrapping A particular problem in this respect is bootstrapping: whenever one wants to verify programs in some programming language X, one will need a formal semantics for programming language X. In order to ensure that the interpreter for X implements this semantics to the same level of confidence than the verification itself, one has to formally verify the interpreter. Now, the interpreter is usually written in some other programming language Y. In order to verify it, we hence need a formal semantics for Y and – in order to assure that this does not just move to problem to the next language – we would need to assure that the interpreter for Y is faithfully implementing the semantics for Y, and so on.

This problem is sometimes described as "turtles all the way down"[17] and illustrates that there cannot be a 100% guarantee. At some point someone must have implemented his verifiable programming language in a non-verifiable one, because otherwise there would not be any verifiable languages. Hence there will always be a loophole in any correctness proof.

The only thing we can do is to make this loophole smaller though additional manual effort (verifying interpreters or compilers), which is why it is important to know the level of confidence expected from the system in order to choose the amount of effort one should invest in developing it. In practice, static analysis and verification can provide meaningful additional confidence for program correctness (just as testing or reviews can). Verification is not a panacea and should not be trusted blindly, but rather seen as a part of a comprehensive quality assurance approach.

2.13 Challenges

There is no free lunch, not even in formal verification. In this section we describe some of the challenges involved in using formal methods, from the perspective of the end user. Whether their use makes sense despite the challenges ultimately depends on the correctness guarantees needed for a given DSL. Additional challenges might arise for the developer of the DSL that integrates verification; see Sec. 2.11.

Designing for Verifiability Formal methods have limits to scalability – both in terms of the users's ability to write and understand specifications (remember: they have to be "obviously" correct) and in terms of the verification tool's scalability. We have mentioned before that the solution to this challenge is modular/hierarchical verification. However, this only works if one has reasonably small and well-defined modules. The modules and their boundaries have to be designed explicitly. It is very hard to take an existing spaghetti-system and try to verify it. So, similar to test-driven development, a system has to be designed to be verifiable – on the other hand, building a system that *is* verifiable leads to a well-defined system.

Leakiness of Abstractions A core idea of the approach described in this tutorial is to "hide" the verification tool and the involved formalisms behind the DSL. However, sometimes the verification tool requires additional configuration parameters (in addition to the program and the specification). These might be hard to set correctly. For example, for bounded model checking, one has to specify the bounds (i.e., the number of steps that should be considered). If the number is too low, a property violation might not be detected. If the number is too high, the analysis runs (unnecessarily) long. For the end user, finding good values for such parameters can be a challenge. The parameters cannot (always) be set automatically as part of the transformation of the

[17] `https://en.wikipedia.org/wiki/Turtles_all_the_way_down`

DSL to the verification tool (cf. Fig. 3) because there are no obvious algorithms to derive them from the program.

If the parameters affect performance, one approach at automating the process is to use relatively small values in the IDE when the verification runs interactively (low soundness; errors might not be found!), but then run the same analysis with bigger parameter values (higher soundness; more errors can be found) as part of the continuous integration server (one might receive an error from the server even though the local verification runs fine).

Non-Linearity Many verification tools are based on heuristics. This means that, based on the structure of the to-be-verified program and/or the properties, they "guess" or "estimate" the right approach to the verification. This means that, sometimes, a small change to the model or property can have a big impact on the performance (and if used with timeouts, success) of a verification.

This situation is known from relational databases, where, depending on the specific formulation of the SQL query, the performance can vary significantly. The reason is that query optimizers are also often heuristics-based.

2.14 Further Reading

While mention specific references throughout the booklet, this paragraph mentions a couple of general recommended material. First, if you are somebody who learns from listening, you might want to check out the omega tau episode on specification and proof with Benjamin Pierce [46]. A comprehensive (and free) book on static program analysis is the one by Møller and Schwartzbach [30]. Another one is the book by Nielson et al. [32] mentioned before.

3. Type Checking

3.1 A Recap of Basic Types

The purpose of many static analyses is to associate some value – that represents the result of the analysis – with a program node. The value might be constructed through a more or less sophisticated analysis. Type checking is one of the simpler ways of analyzing a program; the value associated with the program node is called the type, and it represents the data this program node can take (e.g., the value `short int` means that the node to which the value is assigned can hold numeric integer data from -127 to 127). Type checking is a value analysis, so only expressions, i.e., program nodes that represent a value at runtime, can have types[18]. The rules for determining the types are typically one of the following:

- Fixed types for certain language concepts. For example, the number literal `1` always has type `int`, and the type of `int` is also always `int`[19]. Sometimes the type might depend on internal structure: for example, a number literal with a dot in its value (as in `33.33`) might be typed as `float` instead of `int`.
- Types may be derived from other types. For example the type of a variable reference is typically the type of the referenced variable. More complicated type derivations are also common. For example, the type of `+` in `3 + 2.5` is `float`, i.e., the more general of the two argument types. To determine which is more general, a type system has to be able to describe subtyping rules, such as `int is-subtype-of float`.
- Types may depend on the program structure, i.e., the location of the to-by-typed node inside the AST. For example, a variable reference that points to a variable that has type `option<int>` also has type `option<int>` in general (as discussed above). However, if the reference occurs under a node that ensures that the option is not `none` (e.g., `if x then x + 1 else 0`, the type could be `int`.

Languages with static type checking can support type inference. This means that a type does not have to be specified explicitly by the user. So instead of writing `var int x = 10;` the user can just write `var x = 10;` and the type system infers the type of `x` from its init value 10. Explicitly given types can be seen as a specification against which types that are computed from other program structures are checked.

Type inference is useful if types become more complicated. For example, it is nice if a type `map<string,`

[18] For type system implementation reasons, other program nodes, such as declarations, might also have a type, even though they do not hold values at runtime.

[19] To simplify the implementation of the type system, it is often useful to also assign a type system type to AST nodes that represent types.

`list<Person>>` does not have to be written down on the left[20] and the right side of a variable declaration. However, most languages that support type inference still require explicit types for function arguments, so this complicated type still has to be repeated a lot. A better/complementary way to solve the repetition problem might thus be to support `typedef`s where a complex type like the one above is given a short alias (`PersonMapList` in this case) that then can be used throughout the program.

3.2 Structured Types

Structured types are types where the type node itself has structure; in contrast, an `int` type is *not* structured, it is just this single token `int`, and all `int`s are exchangeable.

Types with a Declaration The simplest kind of structure in a type is a reference to a declaration. For example, a type `Point` that types all values whose structure conforms to `record Point {x: int, y: int}` is typically modeled as a `RecordType` node which points to the declaration it represents (the record `Point` in this case). Usually, all `RecordType` instances that point to the same `record` declaration represent the same type. Such types are typically covariant (see next paragraph).

Parametrized Types Types can be parametrized with other types, a mechanism also known as generics. Theor most common use is collections: a list of integers might have type `list<int>`. Another example are option types, such as the `option<int>` mentioned above, which represents the fact that the variable can either hold an `int`, or nothing at all (often called `none`).

Subtyping rules for parametrized types are less obvious; for example, a `list<T>` might or might not be a subtype of `list<S>` if `T` is a subtype of `S`. If this is the case, then it is said that the `list` type is covariant with regards to its parameter.

Parametrized types are more precise than non-parametrized types. A `list<int>` can reject adding any values that are not `int`s, something a general `list` type could not do. Program nodes may transform that type, for example `list(1,2,3).select(it.toString)` will by typed as `list<string>`.

Dependent Types The parameters for parametrized types mentioned above are *other types*. In dependent types, these parameters can be arbitrary values [50]. Typically, these values change based on the program structure. For example, consider a number type `number[min|max]`, where `min` and `max` are integer values that determine the range of values allowed for the type. If two instances of such a type are added `c = a + b`, then the type of `c` is a `number[min_a + min_b|max_a + max_b]`. Thus, the type *depends* on the program and the control flow (the type annotates a value and not an AST node). The typing rule must express this:

```
concept PlusExpression
  left operand type:  NumberType
  right operand type: NumberType
type (operation, leftOpType, rightOpType) -> node<> {
  node<NumberType> res = new node<NumberType>();
  res.setRange(
    InfHelper.add(leftOpType.lowerBound(),
                  rightOpType.lowerBound()),
    InfHelper.add(leftOpType.upperBound(),
                  rightOpType.upperBound())
  );
  return res;
}
```

As you can see, this typing rule is specific for `PlusExpression` because it has to compute the ranges in a way that is specific to the semantics of plus. Similarly specific rules are required for all other numerical operators. This is significantly more effort than the typing rules for regular, non-dependent number types, where one can write a generic typing rule along the following lines:

```
concept PlusExpression | MinusExpression |
        MultiExpression | DivExpression
  left operand type:  NumberType
  right operand type: NumberType
type (operation, leftOpType, rightOpType) -> node<> {
  return computeCommonSuperTypeOf(leftOpType, rightOpType)
}
```

We have implemented these kinds of number types in the KernelF language, a functional core language that is embedded in DSLs.

Another obvious use case for dependent types is list types, because it might know its size `list<T, int>`. A list literal `list(1,2,3)` would have type `list<int,3>`. If an element is added to that list, the resulting list has type `list<int,4>`. If, through this mechanism, a value has type `list<T,0>`, we can statically prevent the call to the `take` operation, which would be invalid because the list is empty, and nothing can be taken from it.

Implementing dependent types correctly and throughout a language is a major undertaking and requires lots of sophistication in the type system. In fact, dependent types are one way of implementing theorem proving [8]. Dependent types have many more uses, though. For example, a code generator that wants to select the most efficient representation for a particular implementation variable can exploit the range of the values that are possible at any given location in the program, and thus derive the required size for the implementation type. Also, by statically tracking the possible values for variables, one can verify aspects of the correctness of a program. Consider the following code, which applies a correction factor to a measured blood pressure:

```
type BloodPressure: number[60|200]  // range from 60 to 200
fun measureBP(): BloodPressure = {...}
fun correctedBP(): BloodPressure = measuredBP() * 1.1
```

This code is invalid because the range of values for `correctedBP` cannot fit into the range defined by

[20] The IDE can still show the type if the user requests to see it.

```
derived unit mps = m * s^-1 for speed
convertible unit kmh for speed
conversion kmh -> mps = val * 0.27

#constant MAX_SPEED = convert[100 kmh -> mps];

int8/mps/ calculateSpeed(int8/m/ length, int8/s/ time) {
  int8/mps/ s = length / time;
  if (s > MAX_SPEED) {
    s = MAX_SPEED;
  } if
  return s;
} calculateSpeed (function)
```

Figure 4. Units in mbeddr C.

`BloodPressure` type (because of the multiplication with `1.1`). A correct program would use two different types for the measured and corrected blood pressure.

3.3 Annotated Types

Types can carry additional information in addition to the set of values an expression can take, at least not in a traditional way. We discuss units and tags as examples.

Units mbeddr [47] C supports physical units for C types, as shown in Fig. 4. The seven SI units are predefined. New units can be derived from basic SI units or other, previously derived units. When types are checked (e.g., in assignments) the types are normalized before they are compared. When values are calculated, for example, using addition or multiplication, the units are calculated as well. These two features make the the assignment at the beginning of `calculateSpeed` in Fig. 4 correct in terms of types; if the two values would be added instead of multiplied, the type checker would report an error. Note that the type calculations and checks have no impact on the *data* type, i.e., the set of values a variable can take: they can be seen as an additional, to some degree orthogonal check performed by the type system.[21]

This clear separation does not hold for convertible units; those represent the same physical quantity as another unit, but use a different numeric scale (e.g., km/h vs. m/s or $°F$ vs. $°C$). Here, conversion expressions can be associated with the unit, and the `convert` expression can be used to implicitly invoke those conversions.

Type Tags A type tag is enum-style information attached to the type, that is tracked and checked by the type system of the KernelF embeddable functional language. Consider a web application that processes data entered by the user. A function `process(txt: string)` may be defined to handle the data entered by the user. To ensure that `txt` does not contain executable code (cf. code-injection attacks), the string has to be sanitized. Until this has happened, the data must be considered tainted (and, for example, may not be stored in a database or rendered on a web page). Type tags can be used to ensure that a function can only work with sanitized strings:

```
// returns an arbitrary string
fun getData(url: string) : string { "data" }
// accepts a string that must be marked as sanitized
fun storeInDB(data: string<sanitized>) : boolean = ...
...
// v is a regular string
val v = getData("http://voelter.de")
// trying to pass it storeInDB fails because it
// does not have the sanitized tag
val invalid = storeInDB(v) // error
// sanitize is a special operator that cleans up the string,
// and them marks it as sanitized; passing to storeInDB works
val valid = storeInDB(sanitize[v])
```

The `sanitized` tag is an example of a unary tag. A type can be marked to have the tag (`<tag>`), to *not* have the tag (`<!tag>`), or to be unspecified. The tag definition determines the type compatibility rules between those three options. For `sanitized`, a type with no specification corresponds to `<!sanitized>`; in other words, if we don't know, we cannot assume the string has been sanitized.

In addition, the system supports n-ary tags as well. They define a set of tag values (e.g., `confidential, secret, topsecret`) with an ordering between them (e.g., `confidential < secret < topsecret`). The type checking for tags takes this ordering into account, as is illustrated by the code below:

```
val somethingUnclassified : string                = "hello"
val somethingConfidential : string<confidential> = "hello"
val somethingSecret       : string<secret>       = "hello"
val somethingTopSecret    : string<topsecret>    = "hello"

fun publish(data: string) = ...
val p1 = publish(somethingUnclassified)
val p2 = publish(somethingConfidential) // ERROR
val p3 = publish(somethingSecret)       // ERROR
val p4 = publish(somethingTopSecret)    // ERROR

fun putIntoCIAArchive(data: string<confidential+>) = ...
val a1 = putIntoCIAArchive(somethingUnclassified) // ERROR
val a2 = putIntoCIAArchive(somethingConfidential)
val a3 = putIntoCIAArchive(somethingTopSecret)
val a4 = putIntoCIAArchive(somethingSecret)

fun tellANavyGeneral(data: string<secret->) = ...
val g1 = tellANavyGeneral(somethingConfidential)
val g2 = tellANavyGeneral(somethingSecret)
val g3 = tellANavyGeneral(somethingTopSecret) // ERROR
val g4 = tellANavyGeneral(somethingUnclassified)
```

3.4 Type Checking vs. Dataflow Analysis

Let us revisit the dependent type for lists that encodes the size of the list in the type. The situtation can become quite complicated. Consider the following example:

```
fun addOrNot(l: list<int, SIZE>, v: option<int>) =
  if v then l.add(v) else l
```

[21] In the plain C code generated by mbeddr for downstream compilation, the units do not show up. They are removed during generation.

We have two problems. First, the calculated type of the function depends on the argument `v`: is it an actual integer or `none`? In this example it is relatively simple to find out (size increases by one if `v` is an integer), but more generally, we have to "execute" the program body to find out how the type may change. If the function contains a loop, or calls itself recursively, this becomes very complicated.

The second problem is that the type if `addOrNot` is either `list<int, SIZE>` or `list<int, SIZE + 1>` depending on the value of `v`. So we have to either allow the type system to work with several types,

```
fun addOrNot(l: list<int, SIZE>, v: option<int>):
   list<int, SIZE> | list<int, SIZE+1> = {..}
```

or alternatively work with more general predicates (types that contain predicates are called refinement types [11])

```
fun addOrNot(l: list<int, SIZE>, v: option<int>):
   list<int, RESSIZE> where SIZE <= RESSIZE <= SIZE+1 = {..}
```

It becomes clear from these examples that types can themselves become arbitrarily complex "programs" and type inference to automatically compute them is absolutely essential. A third option is to not type the function itself, but just type the calls:

```
val l: list<int,3> = list(1, 2, 3)
val biggerList = addOrNot(l, 4);
val sameList   = addOrNot(l, none);
```

The first call to `addOrNot` would be typed as `list<int,4>` whereas the second call would be `list<int,3>`. Since, in a functional language, a value never changes, and cannot be reassigned, this approach can work in such a context. Once we allow side effects, however, the value, and hence, potentially, the type, of a variable may change:

```
var i = 0
// value is 0, type may be number[0|0]

i++
// value of i is 1, type may be number[1|1]

if (someBool) i++;
// value of i is 1 or 2, type may be number[1|2]

if (someBool) i = i + 8;
// value of i is 1, 2, 9 or 10.
// type is either number[1|10] or, more precisely
// number[1|2] || number[9|10]
```

At this point, we essentially need abstract interpretation and data flow analysis in the context of the type system. We discuss those in the next section. However, even a straight forward implementation of some dependent types can make sense in a program. For example, we have used a functional expression language with `number[min|max]` types successfully in several projects. It simplifies the situation, for example, by going to `number[-inf|inf]` (i.e., an unrestricted number type) as soon as recursion or loops are encountered. The type system still catches many relevant bugs even though it is not sound, i.e., it does not guarantee to catch *all* bugs.

4. Abstract Interpretation

4.1 Interpreters and Program Analyses

Interpreters An interpreter is a meta program that executes a subject program. The interpreter understands the semantics of the subject language and implements it faithfully, including the semantics of the operators, branching, and conditionals, but also the data types: their size limits and overflow behavior, if any, are known to the interpreter. An interpreter can also be seen as an operational specification/implementation of the semantics of a language. Consider the following code snippets as examples:

```
void f1() {
  int a, b, c, d;
  a = input;
  b = input;
  if (input > 0) then {
    c = a + b;
  } else {
    c = a * a;
  }
  d = a / c;
}
```

```
int f2(int a) {
  int b = a;
  while (b < 10) {
    b++;
  }
  return b;
}
```

During the interpretation of function `f1`, the interpreter first allocates some memory for the variables `a` through `d`. Then it stores the values acquired from the user (via the `input` command) in `a` and `b`, respectively. Next, it retrieves more input, and, based on the value of that input, it evaluates either the `then` or the `else` branch of the `if` statement, assigning a value to the variable `c`. Finally, it assigns the result of the division to `d`.

For `f2`, the interpreter starts by allocating memory for `b` and assign the value of the argument `a` to it. Then the interpreter keeps incrementing the value of `b` until it reaches 10. However, it may be the case that `a` was already bigger than 10, and, in this case, the loop body is not executed at all. Based on these observations, we can conclude that the value returned by the `f2` function is either 10 or `a`, depending on which one is bigger.

Abstract Syntax Tree Interpreters typically work on the abstract syntax tree (AST) – they can be seen as a method `eval` on each AST node. ASTs represent the structure of a program. As the name suggests, this data structure has a dominant containment hierarchy: a node may have several children, and every node has exactly one parent (except the root, which has none). In Fig. 5 (A), the solid edges represent containment. In addition, the AST also has reference edges[22], represented as dotted lines. Fig. 5 (A) shows the AST of the example function `f2` (which is repeated in Fig. 5 (E)).

In addition to its use in interpreters, the AST is used for all structural analysis (cardinalities, name uniqueness,

[22] Technically, this makes the data structure a graph, but because of the dominant containment, it is typically still referred to as a tree.

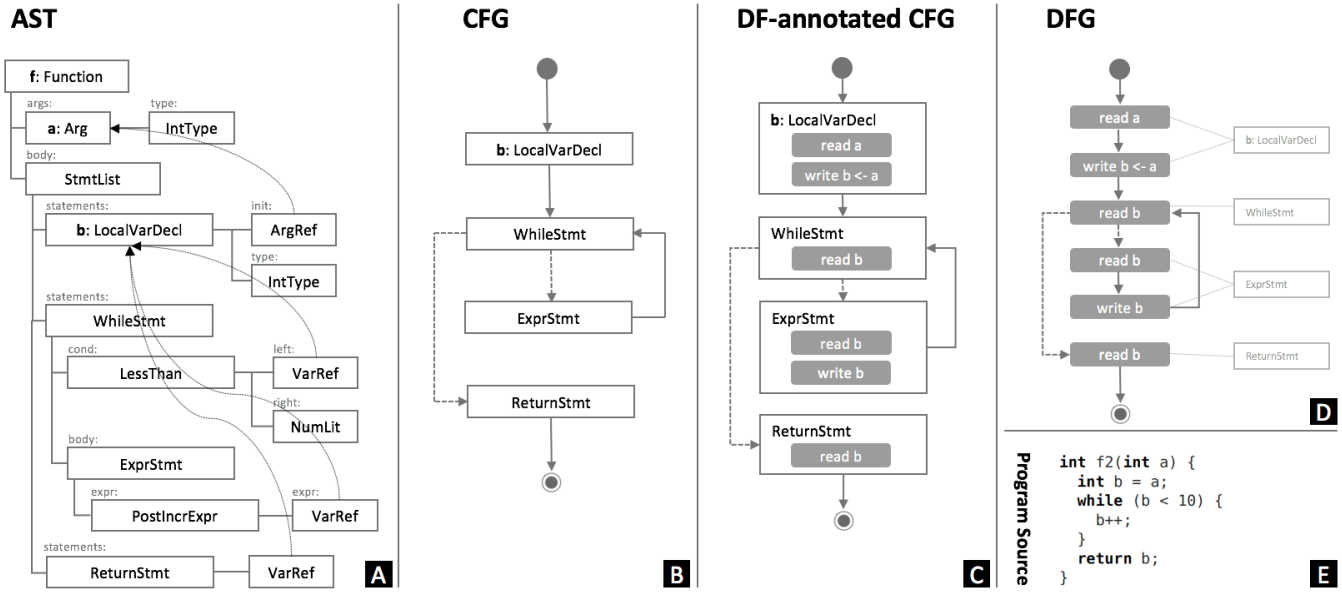

Figure 5. The various data structures used for program analysis. For the example program is shown in (E), (A) shows its AST, (B) is its control flow graph, (C) is the control flow graph annotated with data flow instructions, and (D) collapses these annotations to obtain a data flow graph.

or constraints that limit `assert` statements to test cases) as well as for type checking (e.g., the type of the `init` expression in the `LocalVarDecl` must be the same or a subtype of the declared `type`).

Note that there is also a concrete syntax tree (CST), or parse tree, that includes layout, formatting and other concrete syntax specifics. However, since the CST is only relevant in parser-based systems, and because it is irrelevant to the vast majority of analyses, we do not discuss it here.

Abstract Interpretation We now focus on function `f1`, and assume that we would like to ensure that no division by zero happens during the execution of the program – which is potentially possible if `c` is zero. In order to analyze whether `c` can be zero, we could use the interpreter to evaluate the code snippet. However, the interpreter can evaluate a program only for a specific set of inputs; however, we are interested whether a division by zero can occur in any of all possible executions of the program. We could run the interpreter for all possible combinations of inputs, but this can quickly become infeasible because of the ranges of the variables and the resulting combinations of values, a problem known as state space explosion. However instead of using the actual data types (with their huge value ranges) we could alternatively use abstract/approximate types, i.e., types that have a smaller set of values. This reduces the size of the state space and makes the evaluation feasible. Of course this might come at the price of generality of the analysis or the precision of the results. The choice of the abstract type (also called *domain* in this context) is crucial because, on the one hand, we want the analysis to be as precise as possible which requires a domain that does not abstract away too much information, but, on the other hand, we want the analysis to be computationally feasible (so that it is useful in an IDE) which requires that the domain abstracts significantly over the actual data types. Resolving this trade-off is one of the main design decisions when implementing an abstract interpreter and is an example of the precision vs. performance trade-off mentioned in the introduction.[23]

So in our example, instead of tracking all possible values in the integer domain, we introduce an abstract domain that just distinguishes between the integers based on their signs. In addition to "replacing" the data types used by the interpreter by introducing an abstract domain, we must also adjust the interpreter behavior for the operators correspondingly.

Such an abstract domain is usually represented as a lattice. A lattice L is partially ordered set where every subset $S \subseteq L$ has a least upper bound (LUB) and a greatest lower bound (GLB). Fig. 6 shows the `sign` lattice (in general, lattices can have more than one level; we show a more complex one later). The partially ordered nature of the lattice elements means that ordering is defined only between some of the elements (e. g., $\bot \leq +$, $\bot \leq \top$, and $+ \leq \top$), whereas other elements, those on the same level in the graphical representation, have no order defined between them (e. g., $+$, $-$, and

[23] Many of the examples and explanations in this section were motivated by the lecture notes on static program analysis by Andreas Møller [30].

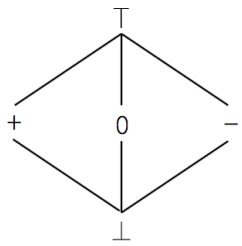

Figure 6. Sign lattice: $\bot$ represents the empty set of integers, $+$ represents the positive integers, $-$ represents the negative integers, 0 represents the singleton set containing the number 0, and $\top$ represents the complete set of integers.

0). Also, finding the LUB of a given set of elements corresponds to finding the least common ancestor in the graphical representation of the lattice. For example, the LUB of the elements $+$ and $-$ is $\top$. Similarly, finding the GLB corresponds to identifying the greatest common descendant in the diagram. The LUB and GLB operations are fundamental operations for lattices and are particularly important in program analyse because of their monotonicity: the repeated application of LUB guarantees that we never go lower in the lattice than where we were before. Similarly, applying GLB to a set of lattice elements will never yield an element that is higher in the lattice than the individual elements themselves. These guarantees are the foundations of fixpoint computations, which are used in analyses; as long as we use sequences of GLB or LUB operations on a lattice, the lattices ensures that a fixpoint will be reached eventually because the computation converges towards one direction in the lattice (and they never "turn back"). Fixpoints, in turn, are important, because they guarantee termination of the analysis.

The benefit of using an abstract interpreter instead of the concrete one is that one evaluation with the former covers a potentially large set of evaluations with the latter. For example, based on the abstract value assignments `a = +, b = +`, the abstract interpreter can derive the assignment `c = +` even without knowing which branch of the `if` statement would be taken. This knowledge about `c` is already enough to find out whether a division by zero can happen; however, there is more to this analysis, and we will revisit it later. To sum up: because the abstract values cover a large portion of the program inputs and execution states, we managed to avoid examining each one of them individually, thus making the evaluation computationally feasible.

Program Analysis As we have explained in Sec. 2.2, program analysis is the process of automatically analyzing the behavior of a subject program with regards to some property (e. g., safety, robustness, liveness) and

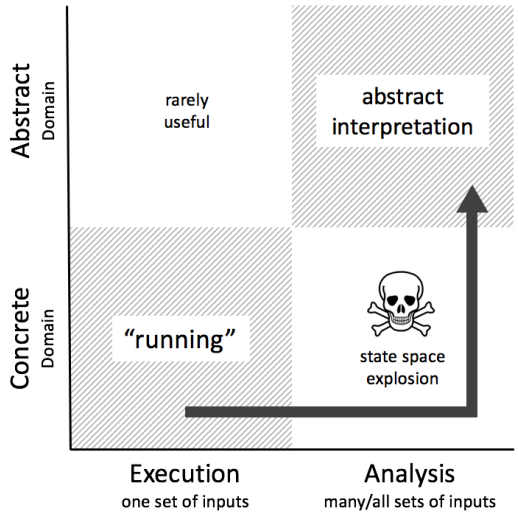

Figure 7. Execution and analysis with abstract and concrete domains. The potential for state space explosion when "running" the program for all possible inputs forces us to use an abstract domain.

"annotating" the program with the information gathered by the analysis. Abstract interpretation is one particular way of performing such an analysis; conceptually, it "runs" the program for many (all) possible inputs; it then maintains the analysis results for each node. The actual analysis uses this mapping; for example, it identifies and marks program locations that perform a division where the divisor is a reference to a variable that can potentially be 0.

To recap, Fig. 7 visualizes the transitioning from simple "execution" to analysis. We run (or execute) the program on a specific input; for that, we use a concrete domain (the data types originally used in the program) and a concrete interpreter. However, an analysis of the subject program requires that we reason about more than one (and potentially all) possible sets of inputs/executions. This comes with a price: namely, the potential for state space explosion. We can tackle this problem with the introduction of an abstract domain, abstracting away from all intricacies of the concrete domains that are irrelevant for the analysis in question. This leads us to abstract interpretation. The combination in the upper left quadrant of Fig. 7 is marked as rarely useful because *executing* the subject program on an abstract domain does not yield better results than executing it on the concrete domain.

Let's revisit our example program and add an analysis on top of the abstract interpreter which marks potential division-by-zero errors in the subject program. The result of the abstract interpretation is shown in the comments in the program below.

```
1 void f1() {
2   int a, b, c, d;
```

```
3    a = input;              // a = ⊤
4    b = input;              // a = b = ⊤
5    if (input > 0) then {
6      c = a + b;            // a = b = c = ⊤
7    } else {
8      c = a * a;            // a = b = c = ⊤
9    }
10   d = a / c;              // a = b = c = d = ⊤
11 }
```

All variables have been analyzed to have the value $\top$, the top value of the lattice. In other words, all variables may have all values from the lattice, and thus, they might be zero. The division-by-zero analysis uses these results and marks statement `10` as erroneous because it is possible that the value of `c` is zero at that program point.

This result may seem over-approximating (because `c` might not be zero for a particular execution), but for an analysis, we typically want to have soundness (and allow false positives) rather than to miss a potential source of an error (and thus allow false negatives). We can get more meaningful results if we enable the interpreter to "understand" more about the values and semantics of a program. For example, instead of using `int` type for `c`, we could use an `uint` type which only has positive values. Alternatively, for example, if the subject language type system does not support `uint`, we can add hints to the program, as exemplified by the `assume` statements in the code below[24]:

```
1  void f1WithAsserts() {
2    int a, b, c, d;
3    a = input;              // a = ⊤
4    assume a > 0;           // a = +
5    b = input;              // a = +, b = ⊤
6    assume b > 0;           // a = +, b = +
7    if (input > 0) then {
8      c = a + b;            // a = b = c = +
9    } else {
10     c = a * a;            // a = b = c = +
11   }
12   d = a / c;              // a = b = c = +, d = +
13 }
```

The `assume` statements, a very simple form of a specification, express that *at this program location* a particular constraint (such as `a > 0`) holds. The analysis now knows that `c = +`, so a division by zero can never occur. It is common for analyses in general purpose languages to be more (or only!) meaningful after annotations are added to code. In the context of DSLs the situation is slightly different, because one can try to design a language that directly expresses the relevant semantics[25].

This example also illustrates the risk associated with specifications: essentially we tell the analysis "shut up and assume what I specify here". If what I specify is wrong, the analysis might draw a faulty conclusion. To remedy this, on can (a) check a set of specifications/assumption for consistency, assuming that a faulty specification would contradict another (correct) one, (b) verify the correctness of the assumption through another analysis, or (c) also check the assumption at runtime, during testing. Pragmatically speaking, we would probably use the latter approach.

Control Flow Graph It is obvious that the analysis above depends on *where* the assumption are added to the program, and in which order the statements in the program execute. To understand the relevant program nodes and the order(s) in which they (can) execute, we use a specialized data structure, the control flow graph (CFG). It represents the possible execution flow(s) of a program. Its nodes are the control flow relevant locations of the program (e.g. statements, the heads/conditions of `if`/ `else if` statements and loops). An edge is present between two nodes `A` and `B` if the control can flow to `B` after the code at `A` has been executed.

The CFG, shown in Fig. 5 (B) is fundamental for program analyses that rely on the ordering between the program locations. For our example `f2` function, the CFG expresses that statements `2` through `4` are executed sequentially, then the loop head is evaluated at statement `5`. Here the execution branches based on the result of the evaluation of the condition, and the control could flow either to statement `6` or to statement `8`. Note that, as a data structure used for analyses, the control flow graph is representative for all possible program executions, that is, it has edges for both `5 -> 6` and `5 -> 8`.

The CFG is typically derived from the AST of the program as exemplified by Fig. 5 for the example function `f2`. This means that the nodes of the CFG are a subset of the AST nodes; typically these are the statements, but also `else if` branches or a `case` of a `switch`. The dotted lines in the CFG represent those edges that are taken conditionally; in this case, the branching appears at the loop head, control either flows to the loop body or to the return statement.

Data Flow Graph So far, the input to the program analyses was the CFG of the subject program where the nodes represent control flow relevant program locations. At these locations, there are instructions (language constructs) *specific* to the subject language, and program analyses would rely on the abstract/concrete interpreter of the *specific* subject language to interpret these instructions. In other words, the analysis is *specific* to the subject language.

A more general approach would be preferable because (1) we might not have (or want to build) an interpreter for every subject language (2) there are several off-the-shelf program analyses that are potentially independent of our subject language and could be be reused. Addi-

[24] We assume unlimited range of the `int` types; otherwise we'd have to consider overflow. Note that analyses that do consider the overflow behavior of a particular language are considerably more complicated (but also more precise) than those that don't. Another trade-off!

[25] This again hints at the synergies between analysis and language engineering.

tionally, (3) handling all of the language constructs of a subject language in the analysis/interpreter is a lot of work, and finally, (4) handling all kinds of language constructs directly in the interpreter/analysis may not even be desirable because the abstract interpreter needs adjustments when an extension of the subject language is activated.

Again, abstractions come to our aid: we abstract from the concrete language constructs of the subject language only their data flow relevant aspects (e.g., read, write, context switch) and annotate the CFG with those. We call this intermediate form the data flow-annotated CFG. Fig. 5 (C) shows an example. We transform this structure one more step to end up with the data flow graph (DFG): we split up the composite program locations that consist of multiple data flow instructions, and we introduce a straight line control flow between these instructions. Otherwise, the DFG faithfully represents the original control flow of the subject program as shown in Fig. 5 (D).

The benefit of this representation is that it abstracts away from irrelevant details of the subject language, and it uses a small amount of generic data flow instructions (as opposed to the potentially large set of subject-language constructs). Abstract interpreters only need to handle these kinds of instructions in their implementation. This way, we can decouple the analyses from the construction of the DFG; each subject language must be able to construct a DFG, but the analyses are then generic over multiple languages. DFGs are the data structure of choice on which data flow-related analyses are performed. This idea is used by several off-the-shelf tools, e.g., Soot [43] and the MPS data flow analyzer. Sec. 4.3 explains the latter in detail.

Fixpoint Acceleration If a subject program contains a loop, and we cannot statically determine how many iterations the loop will run through, then the analysis potentially runs forever – because during an *analysis*, it cannot know the values of variables that lead to loop termination during *execution*. This is impractical, so we have to approximate the loop with some kind of fixpoint computation. A fixpoint computation is an iterative computation, which, after some finite number of iterations, reaches a stable value that never changes again (the fixpoint).

Consider the `f2` example function from before. We want to perform an interval analysis, i.e., we want to find out the value intervals which the variables `a` and (in particular) `b` can take during the execution of the program; again, this might be useful for optimizing the storage space we want to allocate in a to-be-generated lower-level implementation of this program. So the program analysis computes and maintains a range $[min, max]$ for each program node. The corresponding interval lattice is shown in Fig. 8.

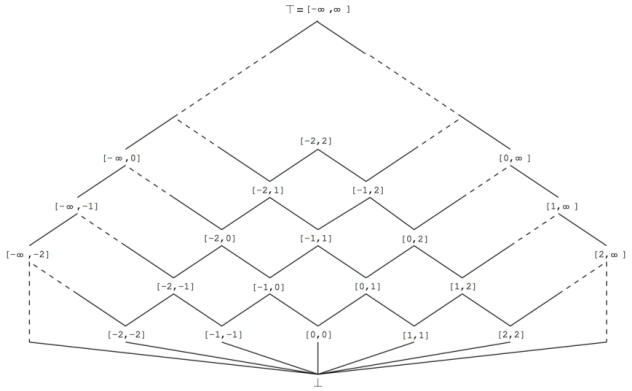

Figure 8. Interval lattice: $\bot$ represents the empty interval, intermediate elements represent intervals where the lower are upper bounds are concrete integers, and $\top$ represents the $(-\infty, +\infty)$ interval.

If `a` were given a value, we could just run the program. However, `a` is unknown, and, once again, we are interested in all possible evaluations of the program. Our key problem here is the loop, which depends on the `b < 10` condition. Let us walk through the analysis.

Initially we assume that the interval analysis ignores the condition of the loop. We start by assigning the unknown value of `a` to `b`. Then, we enter the loop and start incrementing the value of `b` one by one. Here we encounter the aforementioned problem that the interpretation will not terminate because we ignore the loop condition (for now). Specifically, the chain of abstract values associated with `b` at program location 4 is as follows: $[a, a+1], [a, a+2], \ldots, [a, +\infty)$, an infinite number of intervals/steps. This means that the interval analysis would reach a fixpoint in an infinite number of steps – or, one might say, practically does not reach a fixpoint at all (the problem also manifests itself in Fig. 8 because the lattice has infinite height). There are two companion techniques to mitigate this issue: *widening* and *narrowing*. They accelerate the fixpoint computation and make it possible to reach a fixpoint in a finite number of steps.

A widening operator takes a lattice element (in our example, an interval) and returns another element that is located higher in the lattice (potentially $\top$). Narrowing is the opposite as it goes down in the lattice (potentially to $\bot$). Note that these are different operators than the lattice LUB and GLB. LUB and GLB are precise mathematical operations defined for the lattice. They go up or down exactly one level. In constrast, widening and narrowing can jump up or down any number of levels. The definition of those operators does not affect the structure or values of the lattice, but they

make some elements "special", because they are used as targets for the widening/narrowing jumps. The benefit of using widening (and narrowing) is that they speed up the fixpoint computation (potentially from an infinite number of steps to a finite one). However, this also comes with a loss of precision. Thus, different widening and narrowing operators can be defined for a given matrix, depending on the precision vs. performance trade-off required by an analysis.

Widening and narrowing operators are usually defined based on heuristics. For example, one could use all integer numbers appearing in the subject program as intermediate "jump" points for interval boundaries. Once these intermediate points are exhausted as the intervals grow, the widening operator can over-approximate and accelerate the evaluation by widening the intervals to the top element of the lattice. For our example, this would mean that the analysis would not walk up the lattice level by level (and not terminate) but quickly reach the decision that the value of `b` is indeed $[a, +\infty)$.

Widening typically shoots above the actual target, but, for certain subject programs narrowing can potentially make up for some lost precision. In order to demonstrate this technique, we slightly modify the `f2` example function. There is a new local variable `c`, and the loop body also modifies the value of `c`.

```
1  int f2() {
2    int b = 0;
3    int c = 1;
4    while (...) {
5      b++;
6      c = 7;
7      if (...) {
8        c++;
9      }
10   }
11   return b;
12 }
```

Here, the interval analysis would be seeded with the numbers `0`, `1` and `7` as these are the literals that appear in the subject program. The "jump" points for widening in the lattice would then be intervals where the numbers $-\infty$, 0, 1, 7, ∞ occur. The abstract interpreter would go through the following chain of assignments in the loop body: (1) `b = [0,1], c = [1,` ∞`)`, (2) `b = [0,7], c = [1,`∞`)`, and (3) `b = [0,`∞`), c = [1,`∞`)`. Considering this result, there is not much that we can do about the value of `b`, but we could fix the lost precision for `c`. We can refine (or narrow down) this value by a repeated application of the interpreter while disabling widening. This would yield the much better `[1, 8]` interval for `c`.

So far, we have made the assumption that the analysis does not consider the loop condition when computing the result, forcing us to use fixpoint acceleration. Another solution to this issue would be a program analysis where an interval analysis and a control flow analysis (which computes the control flow information) work together in an intertwined manner. That is, the control flow analysis uses the result of the interval analysis to know when to break out of the loop, and the interval analysis keeps interpreting the loop body until the control flow analysis allows it to go into the loop body. However, this improvement is applicable only in certain situations. In case of our original `f2` example function, the problem is that the initial value of `b` is also not known because it comes from the function parameter. For example, if we would know that the initial value of `b` is zero, then the condition `b < 10` would allow the interpretation of the loop body only 10 times. The source of this issue lies in the *local* nature of the analysis: we analyze individual functions in isolation, a so-called intra-procedural analysis. We can improve the precision of the analysis if we consider valid call chains in the subject program and by using the context (e.g., value assignments) of the callers. This leads us to investigating the precision properties of program analyses.

4.2 Sensitivity Properties

Abstract interpretation-based program analyses can be characterized by several sensitivity properties, each incurring different trade-offs regarding precision and performance. Some of these properties also conflict with each other and thus cannot be used together [36].

Flow-sensitivity Flow-sensitivity means that the analysis considers the execution order between statements. In all our previous examples, we used flow-sensitivity because they were based on the CFG or the DFG of the subject program.

To give an example for a flow-insensitive analysis, consider a type checker for a statically-typed programming language where variables are declared with an explicit type. In this case, there is no need for type-checking to be flow-sensitive, as one can just iterate over all assignments ($x = E$) in the code and check that in each case, the type of the expression E is compatible with the type declared for the variable x. The same is true of checks for common programming errors (such as the use of the == operator to compare strings in Java). Such analyses are commonly known as type checks, or just general constraint checks, and rely on the AST. Thus, flow-insensitive analyses should not really be discussed in this chapter on abstract interpretation. We just include it here for completeness.

However, in a dynamically-typed language the situation with type checking is different because types are attached to values rather than variables. There, the types of variables need to be tracked flow-sensitively, as they may change over time. Consider the following program

```
1 x = 1;
2 y = x + 3;        // at this location, x is of type int.
3 x = "foo";
```

```
4 z = x + "bar";  // at this location, x is of type string.
```

Assuming that the operation + cannot be applied to a string and an integer, successfully deriving a type-safety guarantee for this program requires flow-sensitivity.

Flow-sensitive analyses are certainly more complicated to implement and require more memory (storing multiple results for each variable), but usually do not incur heavy performance penalties. Also, every analysis can be made flow-sensitive by transforming the program into static single-assignment form[26] prior to applying the analysis.

Path-sensitivity Path-sensitive analyses are able to distinguish between analysis results for the multiple paths that lead to a particular program location. This is in contrast to path-insensitive analyses which directly merge all results when encountering a merge of control-flows (for instance after a conditional). Consider an interval analysis applied to the following program:

```
1 if (c) {
2   x = -10;
3   y = -2;
4 } else {
5   x = 10;
6   y = 2;
7 }
8 z = x * y;
```

Assuming that we do not have enough knowledge to predict the result of the expression `c`, our analysis is forced to take both branches of the `if` into account. For the true branch, it derives the result $\{x = [-10, -10], y = [-2, -2]\}$ and for the false branch the result $\{x = [10, 10], y = [2, 2]\}$ is calculated. Note that both results are precise. However, a path-insensitive analysis merges the results derived for these paths immediately at a control flow merge point (such as line 8), which results in the significantly less precise result $\{x = [-10, 10], y = [-2, 2]\}$. Thus the final result of our analysis will be $\{x = [-10, 10], y = [-2, 2], z = [-20, 20]\}$.

In contrast, a path-sensitive analysis is able to "delay" the merging of results in order to retain more precision. Hence, the statement $z = x * y$ can be separately analysed for both paths (true-branch (T) and false-branch (F)) that lead to this location, yielding the results $T\{x = [-10, -10], y = [-2, -2], z = [20, 20]\}$ and $F\{x = [10, 10], y = [2, 2], z = [20, 20]\}$. Now, merging them at the end yields $TF\{x = [-10, 10], y = [-2, 2], z = [20, 20]\}$, which as we can see is able to predict the value of the variable z precisely.

Of course, every path-sensitive analysis can only consider some finite amount of paths in the case of loops or recursion. Also, the number of paths through n sequential `if`-statements is 2^n, which is the reason that path-sensitive analyses usually do not scale well with program size.

Intra- vs. Interprocedural All examples given above were *intra*procedural analyses because they analyzed a single function in isolation without considering the possible caller-callee relationships between functions. However, in many cases, the precision of the analyses can be dramatically improved if we also consider these relationships in an *inter*procedural analysis. This requires the construction of an interprocedural DFG that shows which functions can call which other functions and from which program locations. Obviously, performing such an analysis is much more costly because instead of computing one DFG that is specific to the analyzed function, we must construct and analyze an exploded DFG that encodes the complete call graph among functions.

Just like the `assume` statements helped us in the interval analysis example, modularization and stronger contracts can come to our rescue here, as well: if an analysis finds out that, for example, a specific set of values are not allowed for a function's arguments (because, for example, they lead to a division by zero), the function's signature can express this: `f(a: int) where a != 0 {...}`. The analysis for the function will treat this as a constraint (similar to the `assume` shown earlier) and not report the error for the case `a == 0`. On the other hand, the analysis for the calling function *will* report an error if the value does not conform to the constraint of the called function. Thus, in terms of performing the analysis, the two analysis are decoupled; however, they still interact with regards to their results. The idea of giving developers the possibility to express stronger language semantics (with the `assume` at the statement level or via preconditions) for DSLs has a lot of potential because these hints make the analyses more precise while keeping the computational complexity in check. We are exploiting this feature in KernelF.

In languages where the targets of a jump can be computed at runtime, the construction of the interprocedural DFG may require the results of a points-to analysis that determines the potential targets of pointer-typed variables. In languages that have function pointers (such as C), this information potentially determines call targets, as well. However, in order to precisely compute the points-to information, the points-to analysis may require an inter-procedural DFG. This shows the non-trivial nature of DFG construction; it potentially relies on multiple analyses to run in an intertwined manner.

Interprocedurality is in itself not a precision property, it "just" means that an analysis also considers the caller-callee relationships, i.e., an inter-procedural DFG is created. An analysis can then exploit the caller site context to make an analysis more precise. This leads us to the next property: context-sensitivity.

[26] `https://en.wikipedia.org/wiki/Static_single_assignment_form`

Context-sensitivity A context-sensitive analysis is an inter-procedural analysis that considers the calling context (such as the values of arguments) when analyzing a function. Note that this implies analysing a function multiple times when it is called from multiple contexts (with the different arguments values). What specifically comprises the context is specific to the analysis, and it is again a matter of trade-offs which information is included. Consider the following code snippet:

```
1 int f(int a) {
2   return -a;
3 }
4 int g() {
5   return f(-3) / f(4);
6 }
```

An inter-procedural context-insensitive interval analysis of `g` will report a potential division-by-zero error for line 5, because `f(4)` might be zero. This is determined by the following reasoning:

1. the function `f` is called twice, once with `a = -3` and once with `a = 4`. The parameter `a` is hence in the interval $[-3, 4]$.
2. Since `a` is in the interval $[-3, 4]$, the expression `-a` and hence the function `f()` yields a result in the interval $[-4, 3]$.
3. Since a call to `f()` is used as a divisor in line 5 and its result interval $[-4, 3]$ contains 0, there is a potential division-by-zero error in line 5.

This is in contrast to a context-sensitive analysis, which is able to distinguish between the two calls to `f()` and track the different values of the parameter `a` in the context for each call. Hence, such an analysis is able to determine that `f(-3)` will yield a result in the interval $[3, 3]$ and `f(4)` will yield a result in the interval $[-4, -4]$, which does not contain 0.

While context-sensitivity is able to significantly improve the precision of inter-procedural analyses, it comes at the cost of analysing each function multiple times. As mentioned at the beginning, context-sensitive analyses differ significantly in the information they include in their contexts. One common parameter is the length of the call chain suffix considered (often denoted as n-context sensitivity). 1-context sensitivity means that only the callsite of the current function (the last element in the call chain) is considered in the context, which is sufficient for the example above, but would be insufficient if the function `f()` would instead of negating `a` itself, call another function `negate(x)` and pass `a` as an argument to it. In that case distinguishing call sites using 1-context-sensitivity does not help: `negate(x)` has only one callsite (the one in `f()`). However, an analysis with 2-context-sensitivity would be able to distinguish the call chain suffixes `[..., f(-3), negate(-3)]` and `[..., f(4), negate(4)]` as two different calling-contexts for the function `negate()`.

Note that the number of call chain suffixes to consider for each function dramatically increases with the suffix length n. However, researchers have devised clever methods [35] of dealing with this problem and there are even analyses that support call chains of unbounded length while still being able to analyse recursive functions.

Consider the following piece of Java code. Assume that we are interested in the value of `g` after the invocation of `f1` in `f2`, and we use an interval analysis to answer this question.

```
1  public class CS {
2
3    int g = 10;
4
5    public void f1() {
6      g++;
7      if (...) {
8        f1();
9      }
10     g--;
11   }
12
13   public void f2() {
14     f1();
15     // what is the value of g?
16   }
17 }
```

We clearly need interprocedurality to analyze call chains and not just `f1` in isolation. We also need context information, that is, the previous value of `g` when `f1` is called recursively because the analysis needs to increment the previous value by one. In this case, the context would store information about the value assignments. However, there is another critical issue: the handling of the cycle in the inter-procedural DFG introduced by the recursive call. For that, the analysis must also perform "counting"; we enter the `f1` function exactly as many times as we exit it during the recursive evaluation. Because of this, the decrements that happen to `g` will exactly cancel out the increments, leaving the value of `g` to be `10` after the invocation of `f1` in `f2`. So the context must store information about the number of times `f1` was called. With this information, the analysis can filter out unrealizable paths that would represent invalid execution flows, for example, because we would enter `f1` more times than we exit it.

Object-sensitivity is a variant of context-sensitivity for object-oriented programming languages. In this case, the context does not only contain the parameters of the call (and maybe global variables), but additionally the instance variables for the receiver object of the call (such as the member variable `g` in the previous example). Hence, when methods of an object invoke each other, changes to its instance variables are also tracked context-sensitively.

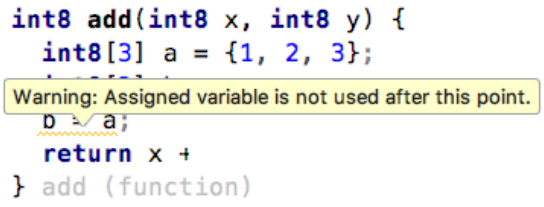

Figure 9. Unused assignment analysis in mbeddr C.

Field-sensitivity Consider the following piece of Java code, and assume that we perform a points-to analysis on this code. Such an analysis derives the possible targets of references in the subject program.

```
1  public class Worker {
2    Integer age;
3    String address;
4  }
5
6  public class Store {
7
8    public void createWorker(Integer theAge, String theAddress)
         {
9    Worker w = new Worker();
10   w.age = theAge;
11   w.address = theAddress;
12   ...
13   }
14 }
```

For this example, a field-*in*sensitive analysis would determine that the potential targets of `w` are the objects represented by `new Worker();` and the `theAge` and `theAddress` objects, even though, the latter two were assigned to fields of `w` and not `w` itself. A field-sensitive analysis considers the fields of a variable `w.age, w.address` separately, whereas a field-insensitive analysis does not. Treating the elements of an array separately can also be regarded as field-sensitivity, but note that this is only possible for a bounded number of elements, as it would otherwise severely impact both memory requirements and performance of the analysis.

4.3 Implementing Dataflow Analyses

Unused Assignment In this paragraph we provide a brief introduction into MPS' data flow analysis by implementing an analysis that finds unused assignments in mbeddr C (for more details on the MPS data flow analysis framework we refer the reader to [38]): if a variable is assigned, but then never again read, this assignment is unnecessary and flagged as a warning. Fig. 9 shows an example.

The implementation of this feature consists of two parts. Part one is the construction of the DFG, which is then used for many different analyses, including the one we discuss here. MPS comes with a DSL for constructing the DFG from the AST where every language construct (i.e., the types of the AST nodes) can contribute code that constructs "its" part of the DFG. For example, for an assignment expression, this data flow builder looks as follows:

```
1  data flow builder for AssignmentExpr {
2    (node)->void {
3      code for node.right
4      write node.left = node.right
5    }
6  }
```

The builder creates a DFG node for the current AST `node`, the assignment expression in this case. It then invokes the builder for the right argument using the `code for` statement. If, for example, the expression on the right is a variable reference, the builder for it contains `read node.var`, which means that the value in the variable is read by that DFG node. Then we express the actual data *flow* with the `write`, i.e., that an assignment means that whatever value is in the right argument now is copied into the variable referred to by the left argument. Data flow builders can become quite a bit more complicated, because, as shown in Fig. 5, all control flow edges, and in particular the conditional ones, have to be constructed.

The second part of this analysis is the actual analyzer that operates on the DFG constructed above. Again, MPS provides a DSL for expressing such analyzers; a screenshot is shown in Fig. 10. The data structure associated with each node is the set of variables `set<node<Var>>` that are initialized at this point in the program. The `forward` direction specifies that the algorithm starts the traversal at the first instruction (a `backward` analysis, such as liveness, would start at the last instruction). The `fun` function builds the aforementioned set of variables for every node in the DFG. When it encounters a `write` instruction in the DFG, it adds the written variable to the set because that variable is now initialized. The `merge` function uses intersection to merge lattice elements. The analysis is sound, because it determines a variable as initialized at a DFG node only if it is initialized on all executions paths that lead to that node (because of the intersection).The actual error marker in source code is created by a validation rule for `Functions` (not shown): it uses the analysis result, iterates over all reads in the function and checks whether the read variable is initialized at the DFG node where the read happens.

Effects Note that the analysis discussed above has to take effects into account. Consider the following assignment:

```
1  int8 add( ... ) {
2    b = g(a, c);
3    return 0;
4  }
```

Based on the above analysis, the assignment to `b` is unnecessary because it will never be read. However, the `g` function might have a side effect, so the whole statement is *not* unnecessary (it might be refactored to just the function call, removing the assignment). A

```
analysis: InitializedVariables
direction: forward
lattice: set<node<Var>>
uses: DefOverride

fun (input, instruction)->set<node<Var>> {
  // handle write (Sec. 2.2)
  // handle map & unmap (Sec. 3.2)
  // handle defInit (Sec. 3.3)
}
merge (inputs)->set<node<Var>> {
  // apply intersection
}
```

Figure 10. Structure of data flow analyzers in MPS.

good error message should take this into account by understanding whether `g` has an effect, or whether it is pure.

To find out whether a function has an effect, further analysis might be necessary. In a purely functional language, functions cannot have effects, so this analysis is unnecessary. However, such a language, in the very end, is also useless. This is why a better approach is to make effects explicit, so that they can be analyzed.

For example, in KernelF, a functional language used as the core of DSLs, a function call has an effect if the called function declares that it can have an effect. Inside a function, one can only call effectful functions (or use effectful primitives) if the function declares an effect (shown by the asterisk):

```
1 int8 computeAverage() {
2   while ( averageUnstable ) {
3     int8 val = measure*();
4   }
5 }
```

```
int8 measure*() {
  // use hardware to measure
  // a value and return it
 }
```

The `measure` function is valid: it declares an effect, an thus it can use effectful primitives (such as accessing hardware). However, `computeAverage` is invalid because it calls an effectful function but does not itself declare an effect.

A simple implementation of the effect-tracking feature relies on the AST alone; no data flow analysis is required. In particular, the following check is required:

```
1  boolean FunctionCall::hasEffect() = this.function.hasEffect()
2  boolean Function::hasEffect() = this.effectFlag
3  check Expression::invalidUseOfEffect = {
4    // the current expression has no effect -> no problem,
       return
5    if ( !this.hasEffect ) return;
6    // find the function that contains the current expression
7    val f = this.ancestor<Function>
8    // The function does not declare an effect,
9    // but the current expression has an effect
10   // (fell through above conditional), so
11   // there is a problem
12   if ( !f.effectFlag )
13     report "cannot use effectful expression in a non-
         effectful function"
14 }
```

However, consider the following case:

```
1 int8 computeAverage() {
2   while ( averageUnstable ) {
3     int8 val = measure*(false);
4   }
5 }
6
```

```
int8 measure(bool measure) {
  if ( measure ) {
    // perform effect
  }
  return -1;
}
```

In this implementation, every call to the `measure` function *might or might not have an effect*, depending on the value of the `measure` argument. Solving this problem requires all of the previously explained sensitivity properties: flow-sensitivity to reason about control flow, interprocedurality to consider whole call chains, and context-sensitivity to, for example, filter out unrealizable paths based on the actual values of arguments and to carry around the effect information. Also, the nature of the analysis suggests that it should be a backward analysis because we carry effects from called functions to callers. In KernelF we use the simpler AST-based analysis, which, while not as precise, is easier to implement. The analysis is sound (it is overly eager in reporting errors), so this is a justifiable approach.

4.4 Incremental Analyses

A problem with all data flow analyses described so far is that they are not incremental; this means that, if the program changes, the derived DFG and all analyses on that DFG are recomputed from scratch. This limits performance and scalability.

The IncA DSL and runtime solves this problem [39]. It supports the definition of efficient incremental program analyses that update their result efficiently as the subject program changes. IncA can express relational analyses, i.e., analyses that create new relations between *existing* AST nodes of the subject program – essentially the construction of the CFG. We implemented several relevant analyses this way (such as control flow analysis, see below), and industrial experience and systematic experiments show that it scales to code bases of industrially relevant size [39]. An extension of IncA, IncA/L also supports the synthesis of new, typically lattice-based data. This is required for the incremental execution of interval analysis or type state analysis.

Relational analyses rely on identifying patterns in an existing graph to relate matched nodes with other matched nodes. Graph patterns are a natural choice, similar to regular expressions that match on strings. Efficient incremental graph pattern matching algorithms and libraries are available [41] for use as a runtime for IncA.

Fig. 11 (B) shows the AST of the program in Fig. 11 (A). Fig. 11 (C) shows the control flow graph (CFG) of that program. Note the cycle in the CFG between statements N2 and N3 that is introduced by the `while` loop in the subject program.

The IncA control flow analysis uses pattern functions to encode relations between AST nodes of the subject

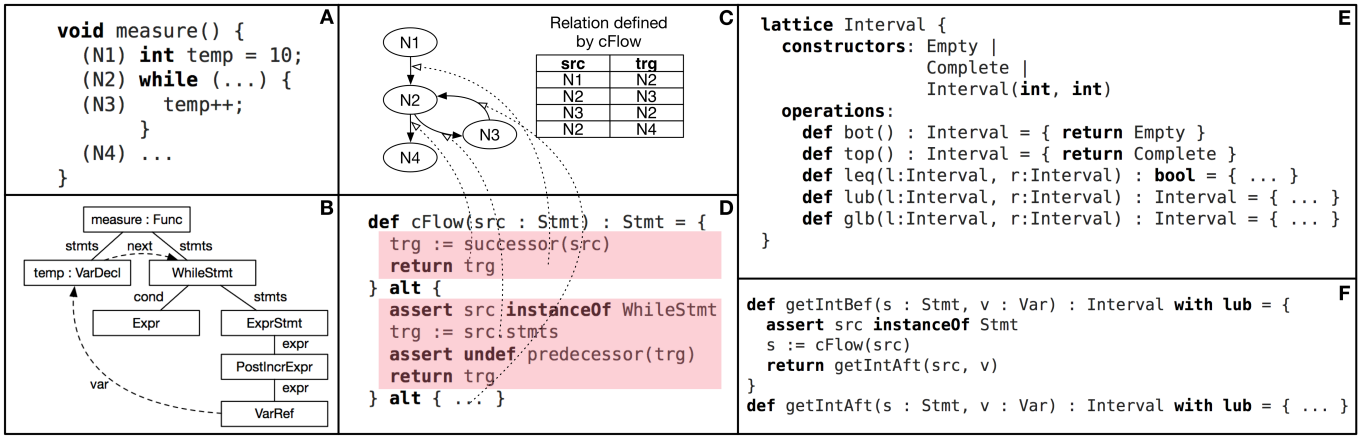

Figure 11. Ingredients of IncA/L program analyses: (A) the analyzed C code snippet, (B) its AST, (C) its CFG, (D) IncA code for control flow analysis, (E) definition of the interval lattice in IncA/L, and (F) IncA/L code for interval analysis. The solid lines in (B) represent containment edges, while the dashed lines represent references to other AST nodes. The dotted lines between (C) and (D) show the alternative body that derives the respective CFG edge.

program – effectively, the edges of the CFG. Fig. 11 (D) shows the `cFlow` function that takes as input a node of type `Stmt` and returns another `Stmt`. A pattern function can have several alternative bodies that each encode a way of obtaining the output(s) from the input(s), thus defining a relation between the program nodes. For example, the second alternative derives the `N2-N3` edge by first navigating to the statements in the body of the `while`, and then returning the statement that has no predecessor (note the `undef` construct for negation) because control would first flow to the first statement in the loop body. In IncA, the result of a program analysis consists of tuples of a relation as shown in Fig. 11 (C).

To support synthesis of analysis data, IncA/L supports lattices, for the reasons introduced earlier. In contrast to the MPS data flow language, the IncA-based solution executes incrementally. Let us look at an interval analysis that derives the potential ranges of values of program variables. Given the code snippet in Fig. 11 (A), assuming that the interval analysis does not know how many times the loop will be executed, it would associate with `temp` the $[10, 10]$ interval at `N1`, $[10, \infty)$ at `N2/N4`, and $[11, \infty)$ at `N3`.

Fig. 11 (E) shows the `Interval` lattice expressed in IncA/L. Fig. 11 (F) shows a part of the interval analysis for C in IncA/L. It consists of two recursively dependent pattern functions `getIntBef` and `getIntAft`. `getIntBef` takes a `Stmt s` and a `Var`, and returns `Interval` that holds the potential range of values for the variable before `s`. `getIntAft` returns the interval after `s`. `getIntBef` uses the previously shown `cFlow` function to obtain the control flow predecessor(s) for `s`, and it returns the interval that was assigned to the given variable after the execution of the predecessor(s) as computed by `getIntAft`. The potential for having multiple CFG predecessors leads us to the requirement for aggregation: instead of tracking individual intervals, we typically want to combine them based on the lattice's `lub` or `glb` operator. For instance, the initial interval for `temp` was $[10, 10]$, and, after the first evaluation of the loop body, we derived a new interval $[11, 11]$. This is propagated back to the loop head through the `N3-N2` CFG edge. We now have to aggregate these two intervals, leading to $[10, 11]$. In IncA/L, the aggregation is controlled by annotations on lattice types as shown in Fig. 11 (F). Both functions use the `lub` annotation which means that the runtime system uses the least upper bound operator to aggregate intervals. This example shows that IncA/L is capable of expressing lattice-based analyses and incrementalize their evaluation.

4.5 Symbolic Execution

Symbolic execution uses the techniques of abstract interpretation to *execute* a program for symbolic value.[27] We go back to a slightly modified example function `f1`:

```
1  void f1() {
2    int a, b, c, d;
3    a = input;
4    b = input;
5    if (a > 0) then {
6      c = a + b;
7    } else {
8      c = a * a;
9    }
10   d = a / c;
11 }
```

When we assign values to variables through the `input` expression in lines 3 and 4, we represent the input as a

[27] Probably because we run the program for *one* value, albeit symbolic, it is called execution and not analysis. Or it's just a strange quirk of history.

symbol, for example $a = \lambda$ and $b = \tau$. As we progress through the CFG, we traverse conditions. For example, in line 6, we know that $\lambda > 0$ and $c = \lambda + \tau$. Otherwise we know that $\lambda \leq 0$. In line 9 we can say that

$$\begin{gathered} a = \lambda \\ b = \tau \\ a > 0 \wedge c = a + b \quad \vee \quad a \leq 0 \wedge c = a * a \\ d = a/c \end{gathered}$$

To answer our original question whether a division by zero can occur we can add another constraint $c = 0$ and solve this set of equations with a constraint solver. If the solver finds a solution (values for λ and τ, it will report it; this means that for this solution c can be 0. If it cannot find a solution, the solver reports `UNSAT`, which means here that no division by zero can ever occur. The encoding in the Z3 solver looks as follows:[28]

```
1  (declare-var a Int)
2  (declare-var b Int)
3  (declare-var c Int)
4  (assert (or (and (> a 0)  (= c (+ a b) ) )
5              (and (<= a 0) (= c (* a a) ) )
6          )
7  )
8  (assert (= c 0))
9  (check-sat)
10 (get-model)
```

The solver reports $a = 5, b = -5, c = 0$ as a valid solution; a division by zero can thus occur. See the next section for details on SMT solving. Note that we have not used symbolic execution in practice, hence this section is shorter and has no practical examples.

Practical Tools In real-world tools, symbolic execution is usually not performed on the CFG. In static symbolic execution (as performed, for example, by KeY [3]) the program is interpreted by rules. Whenever the symbolic execution engine cannot follow a single execution path, execution splits resulting in the symbolic execution tree. In dynamic symbolic execution the program is executed with concrete values. Symbolic values are maintained in parallel. Whenever a different execution path is feasible, execution starts from scratch with different input values to follow that path. Java PathFinder [17] does this with help of its own JVM. In both cases infeasible execution paths are detected by a contradiction in the path condition (the conditions collected along the path).

[28] We have skipped the division itself; we know that we only have to analyze for $c = 0$.

5. SMT Solving

5.1 Introduction

Satisfiability Modulo Theories (SMT) solvers are a category of automated theorem provers that, instead of deciding the validity of (mathematical) theorems directly, focus on solving the equivalent satisfiability problem for logical formulas.[29] For more information on theorem proving in general, see [8].

Hence, given a logical formula f, an SMT solver either

- states that f is satisfiable (`SAT`) and provides a model (an assignment of values for all free variables in f such that f is true under this assignment), or
- states that f is unsatisfiable (`UNSAT`), which means there is no such model. However, in this case, modern SMT solvers are able to provide a proof or other information that helps locating and understanding the problem (unsat Core[30]).

Contrary to SAT solvers, which only support propositional logic, SMT solvers support full first-order logic along with a number of so-called *theories*. This means that, depending on the particular tool and its supported theories, it can also deal, for example, with integer/real/float arithmetics, bit vectors or arrays and lists.

Note, however, that in general, satisfiability of first-order logic with integer arithmetics alone is undecidable, which is why SMT solvers follow a heuristics-based approach. It works amazingly well for most formulas, but for some formulas, SMT solvers will inevitably return the result `UNKNOWN`.

Various SMT solvers exist (such as Alloy [28], Choco [20] or Z3 [13]; we use Z3), and they differ mainly in which theories they support, how well they are able to scale with the problem size, as well as their particular APIs. Contemporary solvers can solve thousands of equations with tens of thousands of variables in very short time spans (sub-seconds to seconds). However, as another consequence of their heuristic approach, a small change to the input formulas may have a significant impact on the solver's performance.

Introductory Example Since SMT solvers take first-order formulas as input, the problem at hand must be encoded as such a formula. We will illustrate this with the decision table in Fig. 12. It calculates whether a recommendation to shift up should be shown to a car's driver, which depends on the car's speed and whether it has a manual or automatic gearshift.

For a decision table to be valid, it must satisfy two criteria. For both dimensions (row headers and column headers), the options must be complete (for all possible values of the inputs, at least one must match) as well as

[29] A logical formula f is valid iff $\neg f$ is unsatisfiable.

[30] For our work with solvers, the unsat core is not relevant.

	mode == *MANUAL*	mode == *AUTO*
speed < 30	false	false
speed > 30	false	true
speed > 40	true	true

Figure 12. An example decision table. Decision tables return a value based on two dimensions (rows, columns) of criteria.

overlap-free (for no combinations of values, more than one option must match).

Let us take a look at the completeness of row headers of our example table. Intuitively, the row headers are complete if one matches for any value of `speed`. In other words, any given value of `speed` must either be less than 30, more than 30 or more than 40. To turn this into a satisfiability problem that can be solved with an SMT solver, we can negate it and ask the solver to find a value for `speed`, such that none of the three expressions is true.[31] The formula sent to the solver would thus be

$$\neg(speed < 30) \wedge \neg(speed > 30) \wedge \neg(speed > 40)$$

... implicitly asking the solver to find a value for `speed` that makes this formula become true. In general, for a decision table with n row headers $r_1, ..., r_n$, the decision table's rows are complete iff the following is *un*satisfiable:

$$\bigwedge_{i=0}^{n} \neg r_i$$

In our example above, the solver is able to satisfy these equations and finds the model $speed == 30$. The row headers are hence not complete, since the case where *speed* is exactly 30 is missing.

The other validity criterion was overlap-freedom: for any particular set of input values, only one of the expressions must be true. The row row headers $r_1, ..., r_n$ of a decision table are overlap-free, iff for $i, j \in \{1, .., n\}$,

$$i \neq j \wedge \bigwedge_{i'=1}^{n} i = i' \Rightarrow r_{i'} \wedge \bigwedge_{j'=1}^{n} j = j' \Rightarrow r_{j'}$$

is unsatisfiable. If it is satisfiable, then the model would indicate a) which expressions (i, j) overlap and b) a value assignment that makes both expressions true. As can be seen from the above example, the expressions that must be passed to the solver can become voluminous, and the various negations can make things unintuitive. It thus makes sense to provide intermediate abstractions based on the observation that many user-relevant problems can be phrased in terms of the following core idioms for a list of boolean expressions $E_1, ..., E_n$

[31] The use of negation to "drive" the solver in a particular direction is typical.

Applicability: Is there a value assignment satisfying all E_i? Examples are any set of boolean expressions, or even a single complex one.

Completeness: For any combination of inputs, does at least one expression E_i match? Examples include conditionals, `switch` statements, `alt`-expressions (see below), decision tables, or transition guards in state machines.

Overlap: For any combination of inputs, does at most one expression E_i match? Examples include any set of Boolean expressions that are *not* ordered, so no two can match any given set of inputs. Often used together with completeness, hence the same examples apply.

Subset: For any $i \in \{1, ..., n\}$, are the values satisfying E_{i+1} a subset of those satisfying E_i? The canonical example is a list of *ordered* decisions, the earlier one must be narrow to not shadow later ones; any kind of subtyping through constraints such as chained typedefs; producer-consumer relationships where the consumer must be able to consume everything the producer creates, or possibly more.

Equality: Are the E_i semantically equivalent, even though they differ structurally (think: DeMorgan laws). Examples include refactorings that simplify expressions.

While these intermediate abstractions are useful in many contexts, this is not an exhaustive list. A tool that uses such intermediate abstractions as part of its architecture must make sure that the list of abstractions is extensible.

5.2 Integration Architecture

Intermediate Language A large variety of end-user relevant SMT-based checks can be encoded as one or more of these basic idioms. It is thus worth making them available as first- class citizens. We call those abstractions the SMT Intermediate Language, or *Solver Language* for short. Fig. 13 depicts the basic idea. Note that the idea of solver-supported languages is not new (we mention Dafny and Rosette later). In particular, Boogie [22] is an intermediate verification language; thus, it has the same goal as our solver language, but is much more sophisticated.

The solver language supports the following concepts: variables with types (Boolean, integer and enum, for now), constraints between those variables as well as checks. A check is a particular question asked to the solver; a check is available for each of the abstractions identified above.

From a user's perspective, the integration of the solver language with the actual solver is generic: it can be used for any user DSL. Only the translation from the user DSL to the solver language has to be implemented specifically

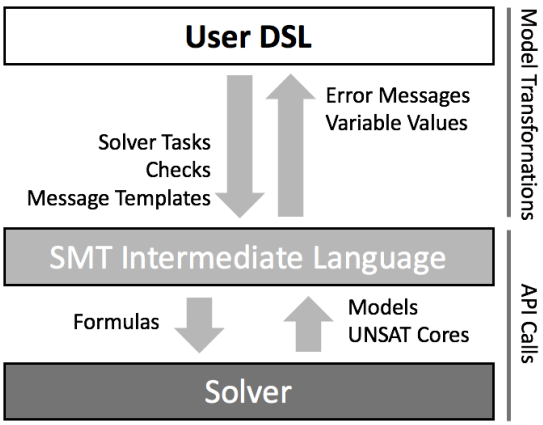

Figure 13. Integration architecture of the SMT solver. We translate DSL constructs to abstractions of an intermediate solver language (using the means of the language workbench) and then use solver-native APIs to communicate the intermediate abstractions to the solver itself.

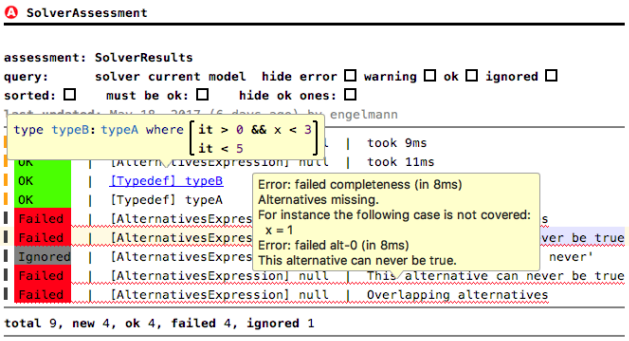

Figure 14. Overview assessment of all solver check. This table shows all solver assessments, whether they are ignored (by an annotation), successful or faulty. All solver checks can be run from this one single place through an intention. To provide context, tooltips show the node in question as well as the detailed error message. Clicking onto the node name navigates to the solved node.

for each DSL. Since the abstraction gap is smaller than a direct low-level encoding of the SMT problem, the effort is reduced. In addition, the step from user DSL to solver language can be implemented as a model-to-model transformation with the means of the language workbench (for example, in MPS), which means that the developer does not have to address the more technical issues such as inter-process communication, timeouts as well as the details of the solver API (we use SMTLIB for the low-level integration [1]).

Simple Example Consider the following program fragment expressed in a DSL; it is called an `alt` expression and can be seen as a one-dimensional decision table: if the condition before the arrow is true, the value after the arrow is the result of the expression.

```
1 fun decide(a: int) = alt| a < 0    =>   1 |
2                         | a == 0   =>   2 |
3                         | a > 0    =>   3 |
```

Similar to decision tables, the three conditions should be checked for completeness and overlap-freedom. Using the intermediate abstraction discussed above, this can be expressed as follows with our solver language:

```
1 variables:
2   a: int
3 constraints:
4   <none>
5 checks:
6   completeness    { a < 0, a == 0, a > 0 }
7   non-overlapping { a < 0, a == 0, a > 0 }
```

Tool Integration Fig. 16 shows the integration of the solver functionality into the end-user DSL and IDE (in this case, MPS). The user starts by writing code that involves a language construct for which solver support is available, in this case the `alt` expression (A). He then receives an info message (the grey underline) as a reminder that a manual check is available (B). This is necessary because, for performance reasons, the solver-based checks are not integrated into the realtime type system checks and must be executed manually. Once the user does this (using `Cmd-Alt-Enter` or mouse actions), he receives an error annotation if the code is faulty (C). Finally, after fixing the problem and reexecuting the solver check, the error disappears (D). Additional support is available for running all solver-based checks in some scope (file, project) in one go and summarizing the results in a table (Fig. 14).

To provide more insight into the reported errors, the solver debugger (Fig. 15) is available. When invoked, it displays the representation of the (relevant part of the) program in the intermediate solver language inlined in the user DSL program[32]. The representation highlights all errors as detected by the intermediate layer. Since that language can also be interacted with, the user is free to play directly with this representation of the problem to better understand it on this layer.

5.3 Transformation to the solver language

The intermediate language is independent of the user DSL and the concrete checks that must be performed there, increasing the potential for reuse. On the flip side, a transformation from user DSL to the intermediate language has to be written for every user DSL. This translation can be separated into two steps.

[32] Thus, for debugging, DSL users have to understand the intermediate language, but not the mapping to the solver itself. Since the intermediate language much more directly expresses the kinds of questions asked to the solver, it is significantly more understandable than the solver code itself.

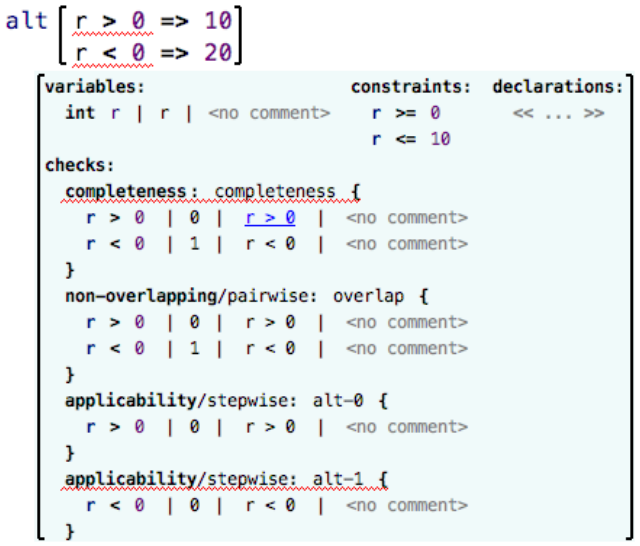

Figure 15. Illustrating a solver-related error using the solver debugger.

Problem-specific Check Construction Depending on the structure of the user program, the transformation has to create the corresponding checks. For example, for the `alt` expression above, this part of the transformation has to

- create a new solver task with a variable for each of the variables in the user program
- create an `applicability` check for all expressions separately
- create a `completeness` check that contains all the options of the `alt` expression
- create an `overlap` check, also containing all the options of the `alt`

If several checks fail, the error message on the user DSL level corresponds to the *first* failing check, so the checks in the solver task should be ordered consciously. In the example we start with the applicability checks of the options of the `alt` expression: if one of them has no values it has to be changed for the `alt` to make sense; checking completeness and overlap with a faulty option is not useful, so we put those first.

Assuming the problem can be represented by a combination of existing solver checks, this step is usually quite straightforward; if no checks exists, the solver language is itself extensible with additional checks as we demonstrate below.

Context Transformation In the previous paragraph we have omitted one crucial issue: the allowed values for each of the variables, i.e., their types and constraints. Obviously, the set of allowed variables affects the outcome of the checks, because, values that are disallowed based on types and constraints cannot be part of the solution the solver potentially finds.

Revisiting Fig. 15, one can see that `r` is typed as an `int` and the constraints section limits its values to [0,10]. This constraint comes from the *context* in which the `alt` expression is used, and in particular, the type of `r`. This type is inferred for the declaration of `r` from its value, which is a call to the function `f` which has an explicit type that limits the values to [0,10]. However, capturing this context is not always so trivial. Consider the following examples:

```
1  type number10: number[0|10]
2  type number5:  number[0|10] where it <= 5
3  fun f(): number10 = 5
4  fun g(): number10 = // something more complex
5  fun h() = // something more complex
6  fun i(): number5 = // something more complex
7
8  val r1 = 10
9  val r2: number[0|10] = 5
10 val r3 = f()
11 val r4 = f() + 10
12 val r5 = g()
13 val r6 = h()
14 val r7 = i()
```

The type of `r1` can be derived from the literal 10 to be `number[10|10]`, leading to an `int` type with a constraint `r1 == 10`. For `r2`, since an explicit type is given, this type should be used even though the actual value implies a stronger constraint. The rationale is that, if the user provides an explicit type, this type should be used for the verification, because, presumable the value of `r2` should be changeable within the bounds of the type without affecting the analysis result. For `r3` the behavior is similar, we take the explicit return type (or the derived one, if no explicit type if given). Note that this type is a user-defined type, so its constraints must be taken into account. For `r4`, the situation is similar to `r3`, except that the inference of the type is more complicated because it must take the arithmetic operations into account. For `r5` the situation is still similar: we do not care about the body of the function, because an explicit type is given, whose implied constraints the tool can readily understand. For `r6` the situation is different: we have to be able to "understand" whatever code is written in the body of the function to derive the type. Depending on the expressivity of the language, this can be non-trivial (and with the current state of our tooling, we cannot do it). For `r7` the situation is apparently better because a type is given. However, this type is a user-defined type that itself contains an expression (that can potentially call into a whole set of functions) to limit the range of the value. So the situation for `r7` is potentially just as bad as for `r6`.

To sum up: deriving the value constraints from the context might require a sophisticated analysis, for example, by translating all of the context language to the

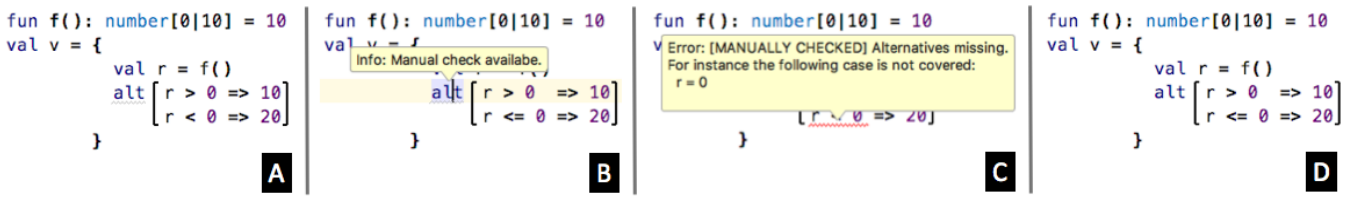

Figure 16. Integration architecture of the solver into a host language from the end-user perspective.

solver, or by querying an interval analysis written with an abstract interpreter.[33]

In KernelF, from which these examples are taken, we have made pragmatic simplifications (for now): the type calculation for number types "widens" to infinity relatively quickly. This limits the precision of the ranges of number types that can be derived from function bodies. Second, we do not take into account invariants on user-defined types, only their declaratively specified ranges (i.e., `number5` would have the same range as `number10`). However, it is feasible to translate complete programs to the solver and to take all implementations of all functions into account (with *some* limitations). The Dafny language developed by Microsoft Research uses this approach [23, 24]. In particular, it statically verifies postconditions of functions against the implementation, given type information, preconditions and `assertions`.

Extensibility The whole point of the intermediate language is that new concepts in existing DSLs, or solver-supportable aspects of new DSLs can make use of the solver integration framework easily; thus, extensibility is very important. This extensibility concerns both the problem-specific check construction and the context transformation aspects introduced above.

For the check construction, one simply implements an MPS generator that maps the domain-specific concept to a solver task. Fig. 17 shows the core of the generator that maps the `alt` expression to a solver task. It works as follows:

- We first create a solver variable for all DSL-level variables used in the `alt` expression. To this end, we call a predefined generator template `variables` to which we pass all expressions from which we want to extract used variables; the template does the rest. In the current case the expression collects all conditions (`variables(node.alternatives.when)`).

[33] Note that the results would be different. Consider a function with the expression `2 * x` where `x` has type `number[1|10]`. Using an interval analysis (e.g., as part of a type system that computes resulting ranges for arithmetic expressions), the result type of the function is the type of `2 * x`, which is `number[2*1|2*10] == number[2|20]`. A translation to the solver would address the problem symbolically, i.e., it would retain the type of the function as `2 * x, 1 <= x <= 10`. As a consequence, a result of 3 is possible for the interval analysis (since 2 <= 3 <= 20) but not from the perspective of the solver because no integer `x` multiplied with 2 will result in 3.

```
variables:
  $CALL$ variables [int x]
constraints:
  $CALL$ constraints [true]
checks:
  $LOOP$ [applicability/stepwise: $ [alt-0] {
            $COPY_SRC$ [true]
          }]
  non-overlapping/pairwise: overlap {
    $LOOP$ [$COPY_SRC$ [true]]
  }
  $IF$ [completeness: completeness {
          $LOOP$ [$COPY_SRC$ [true]]
        }]
```

Figure 17. The generator that creates a solver task from an `alt` expression. Details are discussed in the text.

- We then declare all constraints; similar to variables, we call another existing template `constraints`, passing in the same set of expressions in which we expect DSL-level variables for whose solver-level representation we want to collect constraints.
- We then create an `applicability` check for each of the options separately; the `LOOP` loops over non-`otherwise` options. Inside the check, we essentially duplicate the expressions from the `alt`'s options into the check. MPS' generators ensure that variable references are redirected to the variables defined in the solver task.
- We then generate one `non-overlapping` check; again we essentially duplicate all of the `alt`'s conditions into the body of the check.
- We repeat the same process for the `completeness` check. However, we add this check only if the last of the options is not an `otherwise`, because an otherwise acts as a default option, so the `alt` is by definition complete.

Regarding the context transformation, the challenge is to collect all constraints that apply to a variable. Such constraints can come from the variable's type (e.g., from a number's range), from a `type` definition's constraint

expression, or from a `type`'s base type constraints (which might be yet another `type` definition with a constraint).

To make this extensible, all expressions that are constrained, such as function arguments, local and global values or members, implement an interface `IConstrainedValue`. It has methods to return the name, the (possibly transitive) type, as well as all its constraints. Consider the following example:

```
type posint: number[0|inf]
type age: posInt where it < 110
val ADULT_AGE = 18
type childAge: age where it <= ADULT_AGE
fun f(a: childAge) {}
```

The most complex case is the function `f` where we use an argument `a` of type `childAge`. The constraints for this type can be expressed as

```
var ADULT_AGE: int // type inferred from literal 18
    where ADULT_AGE == 18
var a: int // type of childAge->age->posint->number[0|inf]
    where a <= ADULT_AGE     // childAge
       && 0 <= a <= 100      // age
       && 0 <= a <= inf      // posint
```

This constraint is derived by `&&`-ing all the constraints obtained by climbing the type hierarchy `childAge -> age -> posint`. In particular, the `getConstraints()` method for `type` definitions `and`s its own constraint (if any) with the constraint of its original type, recursively. The `getType()` method follows `type` definitions until they are mapped to a native type. Note that for this to work, all variables or values that are used as part of the definition, such as `ADULT_AGE` must also be converted. Potentially, this can (again) lead to the problem of translating "everything" – which is why, currently, one cannot call other functions from a constraint if solver support should be used.

To sum up, new kinds of expressions can be integrated by implementing `IConstrainedValue` and implementing `getName()`, `getType()` and `getConstraints()` correctly. The reusable generator templates `variables(...)` and `constraints(...)` mentioned above rely on `IConicstrainedValue` and the correct implementation of these methods.

5.4 Some practical experience

KernelF KernelF is a functional core language whose purpose is to be embedded in other DSLs. It contains the usual functional abstractions that can be found in any other functional language, but designed in a way that makes them easy to embed in host DSLs. KernelF is backed by the Z3 solver to detect a number of common programming problems. For example, it contains the `alt` expression, the `type` definitions with constraints, decision tables and decision trees, and the solver provides applicability, overlap, completeness and subset checking where appropriate. KernelF also contains a second form of decision tables that can query over multiple, discrete

```
enum REGION { EU, ASIA, NA, ME }
enum COUNTRY { DE, FR, US, CA, JA }
type cur: number[0|∞]{2}
fun minutePrice(region: REGION, country: COUNTRY, rebated: boolean) =
```

	region	country	rebated	local: cur	longDis: cur
EU-rebated	EU		true	0.80	1.00
EU-non-rebated	EU		false	0.85	1.10
DE	EU	DE , FR	false	0.82	1.05
US	NA	US		0.70	0.75
CA	NA	CA		0.75	0.80
REST				1.00	1.20

Figure 18. A multi-decision table that calculates prices for telephone calls. It uses three query columns and two result columns. The comma-separated alternatives are `or`-ed. If multiple values are returned, a tuple is used.

expressions and return more than one value. Fig. 18 shows an example. As in the previously introduced simpler decision table, this one has to be checked for completeness. However, in contrast to the previous one, this one allows overlap, because the table is evaluated top down. To make sure the wider condition does not shadow the narrower one, narrower expressions have to be appear earlier in the table. This is checked with the `redundancy` primitive check: for example, in Fig. 18, the line introduced by `DE` would be reported as an error because it has to come before the `EU-non-rebated`.

KernelF currently only verifies specific language constructs (those mentioned above), and makes significant simplifications in terms of the context it uses for verification. The details have been described in the Context Transformation paragraph above. In terms of the user interaction, KernelF uses the approach shown in Fig. 16, Fig. 15 and Fig. 14. A core feature of KernelF is the ability to embed it in various host languages and extend the language with concepts that adapt it to that host language in a modular way. We have explained earlier how the solver integration is extensible.

Components In a language for components modeling that also supports user-defined, range-constrained numeric types we have implemented subset checking similar to the `type` definitions in KernelF. However, there was an additional use case. Components have ports that are used to connect instances of those components; see Fig. 19 for an example. A port is associated with a (possibly constrained) data type, but can also specify additional constraints for the allowed values. For the system to be correct, the constraint on the producer side of a connector must allow for a subset of the values that are permitted by the constraint on the consumer side (the consumer must at least be able to consume everything the producer produces). For 1:n ports, this must be true for all consumers. Again, this is a low-hanging fruit for a solver check, but it is not trivial to get right for bigger component-based systems. The language, while similar to KernelF in the structure of

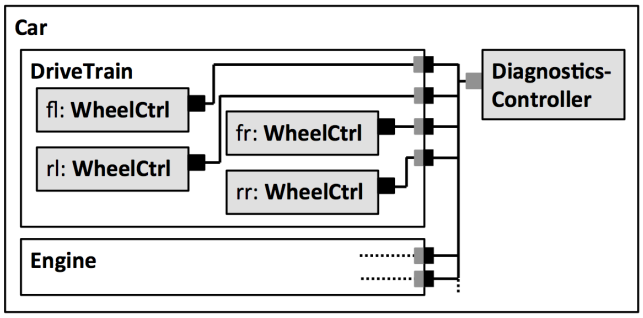

Figure 19. Hierarchically nested components. White boxes are components, grey boxes are instances of (other) components. The small filled boxes are ports. The lines are connectors.

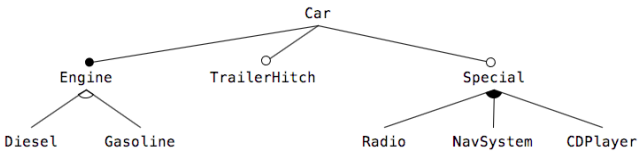

Figure 20. Feature models are used to model the variability of a concept, `Car` in this example. It is used to model (the constraints between) variants in product lines.

type definitions, is nonetheless different, so a separate context transformation had to be implemented.

Medical DSL In a customer project we have been developing a DSL for use in software medical devices. For obvious reasons, the programs written with this DSL should be "as correct as possible", and so an integration of the solver makes sense.[34] In terms of the supported checks, we verified decision tables and decision trees, as usual. In addition, we also checked transitions: the language was based on state machines, and for any given state/event combination, several transitions could be defined with different guards, i.e., different constraints over event parameters or state-local variables. Those had to be checked for completeness and overlap. While this is the exact same check as for an `alt` expression in KernelF, it illustrates the broad applicability of the abstractions identified above, and also highlights the ability of the framework to integrate into different host DSLs.

Variability Variability modeling for product lines relies on features models [12]. A feature describes characteristics/features of the root concept. For example in Fig. 20, the concept `Car` has features `Engine` or `Radio`. The diagrammatic notation expresses constraints between the presence of those features. The filled dot means mandatory (a `Car` must have an `Engine`), a hollow circle means optionality (a `Car` might or might not have a `TrailerHitch`). The hollow arc between features expresses a xor relationship (`Engine` is either `Diesel` or `Gasoline`, but not both) and the filled arch allows one-or-more-of-N, i.e., the logical meaning of "or" (`Special` can be any combination of `Radio`, `NavSystem` and `CDPlayer`, but at least one). In addition, feature models usually allow additional constraints beyond those implied by the tree itself. These are either declarative, as in

```
1 CDPlayer requires  Radio
2 Diesel   conflicts TrailerHitch
```

or might be expressed as Boolean expressions between features directly. More generally, the tree itself is just a visual (and semantic) shortcut for Boolean constraints between the features, and, consequently, every feature model can be translated to Boolean expressions over which reasoning is possible [2, 4]. For example,

```
1 // Engine implies Car; Engine cannot exist without a car
2 // any child->parent relationship results in child => parent
3 Engine => Car
4 // mandatory children also imply parent => child
5 Car => Engine
6 // Exclusives imply the negation of the others
7 Diesel   => !Gasoline
8 Gasoline => !Diesel
```

The first task for the SMT solver is to verify that the constraints implied by the feature model (tree) plus those expressed in addition to the tree still allow some selection of features where any one is used; otherwise that feature is "unreachable" and can be removed (or the constraints corrected).

The primary use of feature models is to define a configuration space, i.e., a set of constrained options from which the user can choose. We call a *configuration* any set of assignments $f_i = true|false$ for all features f_i of a feature model. An SMT solver can now check if such a selection satisfies the constraints expressed by its respective feature model; if not, this configuration is invalid. Essentially, the constraints are conjoined with the true/false assignments and checked for satisfiability.

Ultimately, feature models are used to configure other artifacts (represented by other models that in turn represent software or hardware). Once a configuration is applied to a system, we have constructed a *variant*. There are many ways how this can be done technically, but in almost all cases, parts or fragments of the system model are annotated with so-called presence conditions: a presence condition is a Boolean expression over features of a feature model. For example, a high-performance ECU in the head unit of a car might be annotated with `NavSystem || CDPlayer && Radio`, meaning that this particular model element is present in all variants where the feature `NavSystem` or both `CDPlayer` and `Radio` are selected.

For a system model M, expressed with a language L_M, where model elements are annotated with presence con-

[34] We have built a prototypical integration, but as a consequence of immaturity of the solver integration framework at the time, and project schedule pressure, we did not finalize this integration. However, this is expected to be done for version 2 of the software.

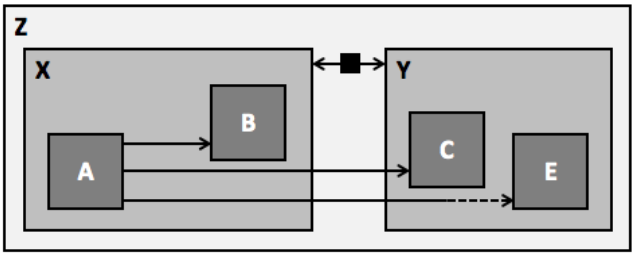

Figure 21. Example program structure used to explain the consistency requirements of variants.

ditions that refer to a feature model FM, the following condition must be true: for all variants allowed by FM (i.e., for all feature selections that are valid with respect to the constraints expressed by FM) all the constraints defined by L_M must hold for M. Consider Fig. 21 as an example. Nesting of boxes represents containment, i.e., syntactic nesting. Arrows represent references, dotted arrows optional references (i.e, they can be `null`, or unset). You can imagine the boxes as components or component instances and the lines as connectors; you can also look at the boxes as syntax nodes in a textual language, e.g., `C` could be a variable declaration and `A` a reference to this variable. Now consider attaching a presence condition to each of the boxes; we denote as P_T the presence condition for a box T. Let us now look at what properties can be verified using an SMT solver.

Valid Tree From the nesting structure, we can immediately see that the presence condition for a nested box is the conjunction of its own presence condition and of all its parents (a syntactically nested box can only be in the program if all its parents are in the program as well). We call this conjunction the *effective* presence condition E_T. The first consistency check we can perform via a solver is to verify that, for every program node T that has a presence condition, the effective presence condition E_T is not empty; in other words, it is possible to select a set of features so that T is present in at least one variant of the system. Otherwise, T would be dead code and could be removed.

Referential Integrity Referential integrity refers to the fact that a reference cannot point to "nothing". This means that the $E_{Ref} \subset E_{Target}$: whenever the reference is in the variant, the target must be in the variant as well. For optional references, i.e., those that are allowed to be `null`, this constraint does not hold. For bidirectional references the two effective presence conditions have to be equivalent. Both equivalence and subset are among the idiomatic solver checks and are thus easy to check.

Other Structural Constraints The above constraints can be derived automatically from the structure of the tree, no special definitions are necessary. However, there are typically other constraints in a language. For example, in a language that describes software deployment onto a set of hardware modules, there might be constraints as to how many software components can be deployed to a specific hardware module (we discuss this example below). If this is described via constraints[35], then those constraints can be taken into account as well and checked against the presence conditions and the feature model.

Types More generally, *any* constraint expressed by the language, and in particular, typing rules, if described as constraints, can be taken in account when verifying the correctness of the presence conditions. However, types are typically not expressed as constraints, and in addition, for real-world sized programs, this will start to lead to performance considerations. So this is still a research topic; we will discuss it below.

Note that all of these check must take the constraints from the feature model into account, i.e., we have to conjoin all the constraints from the feature model with those derived from the program/model structure.

There are various flavours of feature models; one particular extension of the basic formalism uses feature attributes. For example, the `Engine` feature in Fig. 20 might have an attribute `horsepower: int`. Constraints on the feature model might involve those attributes (e.g., `Diesel => Engine.horsepower == 140 || Engine.horsepower == 170`) and variants will specify values for each attribute of each selected feature. Since SMT solvers support integer arithmetics, the attributes and their values in variants can be part of constraints easily.

5.5 Checking vs. Finding Solutions

So far, we have used the solver mostly to check properties of a program (or model): we have submitted a couple of formulas and asked the solver whether they are satisfiable. If they are not, the solver answers `UNSAT`. So, for example, for the following set of equations

$$2 * x == 3 * y$$
$$x + 2 == 5$$

is satisfiable by the model $x = 3, y = 2$. As we can see, finding a model entails assigning values to those variables in the equations that do not yet have a value (i.e., where the constraints still leave some assignments possible). Thus, we can use the solver to *find solutions* for a problem, not just to check the validity of an existing solution. For example, for the feature model example we could ask the solver to "find any variant of the car that

[35] In practice this is often either implicit in the code or checked using procedural/imperative/functional code.

has a `Diesel` engine and a `NavSystem`, if there is one". We will exploit this in the upcoming examples.

5.6 Iterative Solving

Iterative Solving and Optimization Consider the equation $2 * x == 3 * y$, where the variables x and y are of type `number[0|100]`. Let us further assume we want to find the largest possible values for x and y. Here is what we might send to the solver:

$$2 * x == 3 * y$$
$$x \geq 0 \wedge x \leq 100$$
$$y \geq 0 \wedge y \leq 100$$

A valid solution would be $x = 0, y = 0$. It is likely that the solver finds this one, because it represents a kind of extreme case that is likely to be found first by the heuristics-based search. However, these are certainly not the *largest* possible values for x and y to meet the original equation. To drive the solver to less trivial (and in this case, larger) solutions, we restrict the solution space to exclude this (valid, but unwanted) solution by adding the constraints $x > 0$ and $y > 0$, and run the solver again:

$$2 * x == 3 * y$$
$$x \geq 0 \wedge x \leq 100$$
$$y \geq 0 \wedge y \leq 100$$
$$x > 0$$
$$y > 0$$

Next, the solver might find $x = 3, y = 2$; add that to the constraints, and try again:

$$2 * x == 3 * y$$
$$x \geq 0 \wedge x \leq 100$$
$$y \geq 0 \wedge y \leq 100$$
$$x > 0$$
$$x > 0$$
$$x > 3$$
$$y > 2$$

By repeatedly running the solver, and by adding the n-th solution to the constraints before the next run, one can drive the solver to better solutions (in this case, better means bigger, because we want to find a maximum). The iterative process is stopped when either the solver does not find a next solution (`UNSAT`), or when it takes longer than some user-defined timeout. In either case, the last found solution is the best one that could be found by the solver in the allotted time.

This optimization process based on declaring the previous solution invalid can be automated in another intermediate language construct (not done yet).

A more practical example of optimization is the following. Consider again the car example shown in Fig. 20. Let's assume we wanted to find the cheapest variant that contains a `TrailerHitch`. We could do this as follows:

- We assign a `price` attribute/constant to each feature (e.g., `NavSystem.price = 5000`). Conceptually it can be seen as function `int price(Feature)`; SMT solvers typically support such functions directly.
- In order to only count the price of selected features, we can add another function `int effPrice(Feature f) = (f ? f.price : 0)`, which returns 0 as the price for non-selected features.
- We define a variable `totalPrice` that is defined as the sum of all effective prices of all features, i.e., `totalPrice = effPrice(Engine) + effPrice(NavSystem) + ...`.
- We then run the solver iteratively for the variable `totalPrice`, driving it down by consecutively adding `totalPrice < previouslyFoundTotal`.

Test Case Generation Testing functions (and similar "things with argument lists") involves invoking the function with particular values for arguments and then asserting that the result is correct. A core challenge is to come up with a varied set of argument values that "sufficiently" tests the function.

We can use a solver in a similar way as for the optimization case: the subsequent sets of value assignments serve as test vectors. In contrast to the optimization case where we try to "push" the solver in a certain direction by adding constraints that use `arg > prevSolution`, in the test case generation we just use `arg != prevSolution`. However, there are a couple of additional considerations.

If the argument list is unconstrained, i.e., it is just a list of typed arguments, it is relatively easy to find values for those arguments. However, it is also likely that some of these combinations are intentionally invalid: for example, the function implementation might check for this invalid combination and throw an exception. Examples include

```
1  int divide( int a, int b ) {
2    ...
3    if ( b == 0 ) throw InvalidDivident();
4    ...
5  }
6
7  int sendToOneOf( Receiver r1, Receiver r2 ) {
8    ...
9    if ( r1 != null && r2 != null) throw OnlyOneAllowed();
10   ...
11 }
```

The problem is that these checks are buried in the implementation. A better solution is to factor them into declarative preconditions:

```
int divide( int a, int b ) where b != 0 { ... }
int sendToOneOf( Receiver r1, Receiver r2 )
  where r1 != null && r2 == null
     || r1 == null && r2 != null { ... }
```

This way, these constraints can be included in the set of equations sent to the solver, so it is guaranteed that it only finds solutions (test vectors) that respect these preconditions. Of course, a similar check then also needs to be performed at runtime, for example, by generating declarative preconditions into regular `asserts` at the beginning of the function.

Another concern is that, no matter which argument values the solver comes up with, one usually wants to explore the boundaries of the range allowed by the data types, as well as special values such as 0, 1 or -1. So in addition to letting the solver find arbitrary solutions, you can start with some "hardcoded" ones. Note that preconditions are once again helpful, because, if those special values are not allowed, you will just get an `UNSAT` – meaning, you can try them without harm.

A problem with test case generation is that you cannot "generate" the expected results. So by default, you can just run the test and check that the function does not crash. Alternatively, you can of course fill in the expected values manually: this still benefits from the automatic exploration of the inputs. One step better is to also specify postconditions for the functions under test. These can be seen as a kind of implicit test assertions. So if your functions have postconditions, you consider them as an additional approximation if the test succeeded. Obviously, the usefulness of this approach depends on the fidelity of the postconditions. Consider the following example (which you have already seen in the introduction where we have used it as an example of a good specification):

```
list<int> sort( list<int> data )
  post res.size == data.size
       forall e in data | res.contains(e)
       forall i, j: int | i >= 0 && i < res.size &&
                          j >= 0 && j < res.size &&
                          i > j
                          => res[i] >= res[j]
{ ... }
```

This postcondition specifies the complete semantics of the `sort` function (leaving only non-functional concerns open for the implementation): the size of the result must be the same as the size of the input, and the higher-index elements in the list must be bigger or equal to the lower-index ones. If the postcondition is so detailed, the generated test cases, together with checking the postcondition, is all you need for testing[36].

[36] Of course, getting the postcondition correct might be a challenge in itself, as we have outlined in the introduction. So you might want to use a couple of completely manually written test cases as well.

Just as in the case of iterative optimization, you need a criterion when to stop iterating. Again, you can stop when you run into a timeout, but this does not guarantee anything about the number or quality of the generated input vectors. So a better approach is to iterate until you find a minimum number of test vectors. However, you still don't know if those cover the complete implementation. Thus, you can combine test case *generation* with test case *execution*, and measure the coverage of the function, for example, by instrumenting the implementation. You continue iterating until you achieve a specified minimum coverage.

5.7 Advanced Uses of Solvers

Mapping of Sets Consider a typical problem from systems engineering: you have a set of software components as well as a set of hardware nodes. Some pairs of software components have communication relationships through connectors, as shown in Fig. 19. Similarly, some of the hardware nodes have network connections. The problem to be addressed with the solver is: what are valid deployments of software components to hardware nodes. Such a mapping must take into account the obvious constraint that, if two software components communicate, then the nodes onto which the two are deployed must either be the same, or connected (indirectly?) with a network. However, there are typically two additional considerations. First, the bandwidth required by the communication of deployed software components cannot be higher than what is available on the underlying networks. And second, the resource requirements (such as memory or processing power) of the software components deployed on a given hardware node must be lower than what is supplied by the node. In the example below, we only look at the second constraints to keep things simpler. This is an example of a more general problem, where we have

- two sets S and H,
- the type of the elements in S and H are essentially records (i.e., each element has a unique identity plus a list of typed attributes),
- elements within both S and H are potentially linked to each other,
- a number of properties (expressed in a logic specialized to talk about relations, (sets of) nodes, and their attributes as well as links),
- and we are interested in a relation $R \subseteq S \times H$ such that R satisfies all properties.

As usual, the system can then be used to a) check an existing relation, b) generate a new one or complete a partial relation, or c) optimize for a parameter (minimum overall network traffic) through iteration.

The software/hardware example can be concisely expressed with the following DSL (which we have also implemented as a check in the intermediate solver language):

```
record S {
  requiredMemory: int
}

record H {
  availableMemory: int
}

relation communicates S ->  S  // 1:1 communication
relation network      H ->  H  // 1:1 network connections
relation deployed     S *-> H  // allow many S per one H

constraint for h: H {
  forall [s, h] in deployed:
    sum(s.requiredMemory) <= h.availableMemory
}

constraint for [s1, s2]: communicates
  with h1 = deployed(s1)
       h2 = deployed(s2)  {
  h1 == h2 ||  // same node
  network.contains([h1, h2]) // connected throug network
}
```

A given system with three communicating software components and two connected hardware nodes might then be represented as follows. Since the relation `deployed` is unspecified, this is what the solver will compute.

```
// elements of the two sets
s1, s2, s3 : S
h1, h2     : H
// memory attribute values
s1.requiredMemory  = 20
s2.requiredMemory  = 10
s4.requiredMemory  = 40
h1.availableMemory = 50
h2.availableMemory = 20
// communication and networks
communicates s1 -> s2
communicates s1 -> s3
network      h1 -> h2
```

Note that a first check for this model can check whether all required attribute values are set, whether the values have the correct types and whether the cardinalities of the relations (1:1, 1:n) are respected. No solver is required; these are plain old (type) checks on the AST.

Next, we describe the mapping to the solver. We assume the capabilities of Z3, but will spare our readers the Lisp-like SMTLIB syntax. We start with type aliases of `int` that represent hardware nodes and software components.

```
type S: int
type H: int
```

Next, we represent node properties as functions that return the required memory for each software component. We first define the signature, and then `assert` over the outcome of the function call for specific values of `s`. The values of `s` correspond to the enums `s1`, `s2` and `s3`. Essentially, this is a constraint-oriented way of providing a function implementation. Z3 would also support directly defining functions, but the more explicit way we use here leads to UNSAT cores that contain useful information and allow us to inform users about which properties contributed to the problem. The `forall` at the end constrains the function's value for all values except [1..3]; Without it, the solver would be free to choose these values as it sees fit, even outside the range of our "enum".

```
fun requiredMemory(s: S): int
assert requiredMemory(1)  == 20;
assert requiredMemory(2)  == 10;
assert requiredMemory(3)  == 40;
assert forall x: int. (1 < x || x > 3)
  => requiredMemory(x) == 0
```

We use the same approach for the available memory of hardware nodes.

```
fun availableMemory(h: H): int
assert availableMemory(1) == 50;
assert availableMemory(2) == 20;
assert forall x: int. (1 < x || x > 2)
  => availableMemory(x) == 0
```

Next, we model the communication relationships. Note again that we also explicitly model all pairs of components that do *not* communicate, because, if those were missing, the solver would interpret this as a degree of freedom ("if I let these other two components communicate, then I could make this deployment work"). Finally we express with a `forall` that for values outside of [1..3], no communication happens. We then do the same for the network connections between hardware nodes.

```
fun communicates(from: S, to: S): bool;
assert !communicates(1, 1);
assert communicates(1, 2);
assert communicates(1, 3);
assert !communicates(2, 1);
assert !communicates(2, 2);
assert !communicates(2, 3);
assert !communicates(3, 1);
assert !communicates(3, 2);
assert !communicates(3, 3);
assert forall x: S, y: S. (x < 1 || x > 3 || y < 1 || y > 3)
     => !communicates(x,y)

fun network(from: H, to: H): bool;
assert !network(1, 1);
assert network(1, 2);
assert !network(2, 1);
assert !network(2, 2);
assert forall x: H, y: H. (x < 1 || x > 2 || y < 1 || y > 2)
     => !network(x,y)
```

Next up are variables that capture onto which `H` each `S` is deployed. However, we constrain the value range of these variables: they can only be [1..2], because we have only two hardware nodes `h1` and `h2`.

```
var deployed_s1 H
var deployed_s2 H
var deployed_s3 H

assert 0 < deployed_s1 <= 2
assert 0 < deployed_s2 <= 2
assert 0 < deployed_s3 <= 2
```

Because our constraints also require the reverse mapping from `H` to `S`, we also materialize this reverse mapping. Since, potentially, we can deploy each of the three software components onto *the same* hardware node, we have to reserve three "deployment slots" for each hardware node. Each variable `deployed_rev_hX_N` represents the `N`th slot for the hardware node `X`. We also add a value restriction, this time [1..3], because we have three software components. Finally, since each software component can be deployed only once, i.e., can only occupy one deployment slot, we have to require distinctness for those slots.

```
const deployed_rev_h1_1 S
const deployed_rev_h1_2 S
const deployed_rev_h1_3 S
const deployed_rev_h2_1 S
const deployed_rev_h2_2 S
const deployed_rev_h2_3 S

assert 0 <= deployed_rev_h1_1 <= 3
assert 0 <= deployed_rev_h1_2 <= 3
assert 0 <= deployed_rev_h1_3 <= 3
assert 0 <= deployed_rev_h2_1 <= 3
assert 0 <= deployed_rev_h2_2 <= 3
assert 0 <= deployed_rev_h2_3 <= 3

assert deployed_rev_h1_1 == 0 ||
       (deployed_rev_h1_1 != deployed_rev_h1_2 &&
       deployed_rev_h1_1 != deployed_rev_h1_3)
assert deployed_rev_h1_2 == 0 ||
       (deployed_rev_h1_2 != deployed_rev_h1_3 &&
       deployed_rev_h2_1 != deployed_rev_h2_2)
assert deployed_rev_h1_3 == 0 ||
       (deployed_rev_h2_1 != deployed_rev_h2_3 &&
       deployed_rev_h2_2 != deployed_rev_h2_3)
```

Finally, we have to ensure that the two mappings `deployed` and `deployed_rev` represent the two directions of the same mapping (i.e., they are consistent). The assertions below express that if `s1` is deployed on `h1` (i.e., `deployed_s1 == 1`), then one of the three deployment slots for `h1` must point to `s1`.

```
assert deployed_s1 == 1  <=>  (deployed_rev_h1_1 == 1 ||
       deployed_rev_h1_2 == 1 || deployed_rev_h1_3 == 1)
assert deployed_s1 == 2  <=>  (deployed_rev_h2_1 == 1 ||
       deployed_rev_h2_2 == 1 || deployed_rev_h2_3 == 1)
assert deployed_s2 == 1  <=>  (deployed_rev_h1_1 == 2 ||
       deployed_rev_h1_2 == 2 || deployed_rev_h1_3 == 2)
assert deployed_s2 == 2  <=>  (deployed_rev_h2_1 == 2 ||
       deployed_rev_h2_2 == 2 || deployed_rev_h2_3 == 2)
assert deployed_s3 == 1  <=>  (deployed_rev_h1_1 == 3 ||
       deployed_rev_h1_2 == 3 || deployed_rev_h1_3 == 3)
assert deployed_s3 == 2  <=>  (deployed_rev_h2_1 == 3 ||
       deployed_rev_h2_2 == 3 || deployed_rev_h2_3 == 3)
```

This completes the setup; we can now encode our constraints. We start with the constraint for memory use. We sum up the `requiredMemory` for all components that are deployed onto `h1` and `h2`, respectively. Each has to be lower than the `availableMemory` limit for the respective hardware nodes.

```
assert forall h: H.
  h == 1
    => requiredMemory(deployed_rev_h1_1) +
       requiredMemory(deployed_rev_h1_2) +
       requiredMemory(deployed_rev_h1_3)
       <= availableMemory(1)
  && h == 2
    => requiredMemory(deployed_rev_h2_1) +
       requiredMemory(deployed_rev_h2_2) +
       requiredMemory(deployed_rev_h2_3)
       <= availableMemory(2)
```

As the last step, we ensure that communicating components are deployed only onto (directly) connected hardware nodes.

```
assert forall s1: S, s2: S.
                communicates(s1,s2)
       => exists h1: H, h2: H.
            (s1 == 1 => deployed_s1 == h1 &&
             s1 == 2 => deployed_s2 == h1 &&
             s1 == 3 => deployed_s3 == h1) &&
            (s2 == 1 => deployed_s1 == h2 &&
            s2 == 2 => deployed_s2 == h2 &&
            s2 == 3 => deployed_s3 == h2) &&
            h1 == h2 || network(h1,h2)
```

Before we conclude this discussion, let us emphasize two points. First, we have used a very explicit encoding that makes only very little use of solver-level functions. The reason for this is that our experiments have shown that this encoding performs better. The drawback is that the encoding is much more verbose. However, since the code is generated from the more compact representation introduced earlier, this is not a serous problem. Which leads us to the second observation: an intermediate representation, similar to the one introduced above, is clearly helpful, especially when applying the solver to complex problems.

Synthesizing Programs Earlier we have distinguished the use of solvers for checking ("do we get an `UNSAT` or not") and for finding a solution ("tell me a set of values for the variables for which the equations are true"). In both cases, the set of equations (not counting simple value assignments), i.e., those that have been derived from the user DSL, were considered fixed. However, solvers can – to some degree – also be used to come up with, i.e., synthesize, the program that makes a set of equations true. You can imagine this as:

```
int f( int a, int b ) = ??
assert f(1, 2) == 3
assert f(4, 5) == 9
assert f(0, 0) == 0
```

Notice the `??` in the function body: this denotes a "hole" in the program, and the task of the solver is to find an implementation (i.e., a set of equations) that makes all assertions become true. We do not have practical experience with this approach, but for example, Rosette [40], a solver-supported language based on Racket, supports this feature.

The MPS Type System The type system of MPS also relies on a solver. Every language concept C defines a set of type equations, and then, for every instance node c_i, the solver instantiates those equations. A program written with MPS thus leads to a set of equations for which the type checker has to find a solution by solving.

Here are a couple of type equations to illustrate the approach. Note how in addition to equality (the `:==:` operator) the system also supports other relationships, for example `T1 :<=: T2` means that `T1` must be the same or a subtype of `T2`.

```
// the condition of an 'if' must be Boolean
typing rule IfStatement:
  typeof(it.condition) :==: <BooleanType>

// for a local variable, the type of the init expression
// must be the same or a subtype of the type explicitly
// specified for the var
typing rule LocalVarDecl:
  typeof(it.init) :<=: typeof(it.type)
  typeof(it)      :==: typeof(it.type)

// for a list literal list(a,b,c) the type is a ListType(T),
// where T is the supertype of all elements a, b, c
typing rule ListLiteral:
  var T;
  foreach e in it.elements {
    T :>=: typeof(e)
  }
  typeof(it) :==: <ListType(T)>

// define type hierarchy: int is subtype of real
subtyping rule for IntegerType {
  supertype RealType;
}
```

If all types are given in the set of equations, then the MPS type system solver uses the equations for type checking. For example, for the `LocalVarDecl`, the solver checks that the type of the init expression is the same or a subtype of the explicitly specified type (`var x: int = 2` would be ok, but `var x: int = 2.0` would not be). However, if no `type` would be given (as in `var x = 2.0`) then the solver would find a solution for the equations, i.e., it would compute a type. This neatly supports type inference.

The MPS type system is also a good illustration of the *limits* of solver-based specification: realistic languages such as Java, C or KernelF use lots of procedural/functional code as part of the typing rules. We have not managed to express them all declaratively. This might be an indication of our lack of skills, shortcomings in the particular MPS type system language or, as suggested, the *limitations* of solver-based specification.

6. Model Checking

Model checking [10] deals with checking properties of state machines, as they evolve over time. Notice that the term "model checking" is a term of art; It does not refer to the generic notion of "checking some kind of model" for arbitrary problems.

6.1 State Machines and Properties

State Machines A state machine is a finite automaton consisting of transitions between finitely-many discrete system states. Typically, such transitions are triggered by events and guarded by conditions. State machines execute by "being" is one of the states and – when an event occurs – transitioning to a new state by following a transition whose guard condition holds. In addition, entering or leaving a particular state, or following any particular transition may trigger the execution of procedural code blocks called actions.[37] A state machine can be seen as a black box that implements discrete behavior, where the environments sends in events, and the machine changes the environment by actions.

The important characteristic of state machines is that its reactions (its new state and the actions it performs by entering, leaving or transitioning) depends on the *current* state: a machine might reacts differently to the same event, depending on which state it is in. Thus, a state machine can represent (discrete) time as a sequence of states, it can deal with evolving behavior over "time", and it can describe changing system state (not just *the* state, but also values of associated variables). Because it deal with time and state, state machines and the associated model checking are fundamentally different from the logics addressed by the SMT solvers so far.

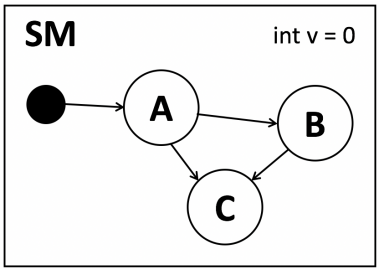

Figure 22. An example state machines used for illustrating properties.

Properties Properties are statements about state machines that have to be true for the state machine to be valid. Since, as mentioned above, state machines encode behavior as it changes over (discrete) time, properties usually quantify over a state machine's execution over time; this can also be seen as execution traces (see

[37] Alternatively those can also be modeled as out events, to make the in/out interface symmetric.

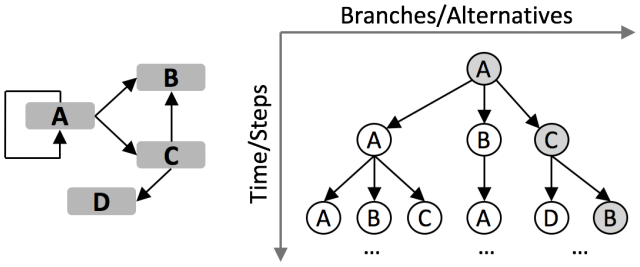

Figure 23. State machines, traces, branches and time. A sequence of states, such as A-C-B, is called an execution trace.

Fig. 23). The kind of logic used for this purpose is called temporal logic. Example properties for the state machine `SM` in Fig. 22 may include:

1. For all possible traces, after `SM` has been in state `A`, it will eventually move into state `B`.
2. For all possible traces, after `SM` has been in state `A`, it will move into state `B` or state `C` in the next step.
3. There exists an trace where, after `SM` has been in state `A`, it will eventually move into state `B`.
4. Before being in state `B`, `SM` has always been in `A`.
5. For all possible traces, `SM` will never reach state `A` after it was in state `C`.
6. Whenever `SM` is in state `A`, a variable `v` will never have a value greater than 10.

Three observations: first, while some of these properties seem trivial as long as the transitions do not have guard conditions, they become less obvious (and hence, more useful to check formally) once guards become involved that use event arguments and variables (such as the `v` in `SM`).

Second, there are two types of quantification involved in these statements: clauses like *for all* and *exists* quantify over execution traces of the machine while clauses like *always*, *never* and *eventually* quantify over time (the sequence of states in an execution trace). Words like `after` and `before` express the temporal nature of a state machine's behavior in terms of the order of states in an execution trace.

Finally, one can distinguish between safety properties (the state machine will *always*/*never* do X) and liveness properties (the state machine will *eventually* do Y). However, from a technical perspective, the same temporal logic is used to express all these properties.

6.2 Temporal Logic

Temporal logics are forms of logic, i.e., reasoning formalisms, that can deal with changing time. In this sense they are more powerful than the propositional logic we have seen before.

LTL and CTL A system that executes linearly, without branches, is said to execute in linear time. A system that can produce different traces depending on (event-triggered) branches is said to execute in branching time. For both forms, temporal logics have been defined: linear-time temporal logic (LTL) and computation tree logic (CTL).

Both temporal logics are parameterized with a logic to reason about states. Typically one uses propositional logic, so one can hence use the usual logical connectives and operators (`!`, `&&`, `||`, `=>`, as well as equality (`==`) and comparison ($<, \leq, \geq, >$) operators within the states alongside the temporal quantifiers to express their temporal relationships. Consider again the last of the example the properties used before: Whenever `SM` is in state `A`, a variable `v` will never have a value greater than 10. The `v > 10` part is a propositional logic expression. However, to make this a useful statement in terms of a statemachine that evolves over time, one has to say something about *when* this proposition holds. For a state machine that can produce different traces (because of branching), we have to quantify both over time (linearly, within one trace) as well as over the traces in the branches. This leads to the following new logical operators available in CTL; the first two quantify over branches, the last four over time. Typically, one combines a branching and a timing operator, as shown in Fig. 24.

A - Always along all possible branches.

E - Exists along at least one path

X - neXt in the next state

F - Future in some future state

G - Globally in all future states

U - Until until some other property becomes true

So if we wanted to express that the property `v == 10` is actually constantly true, we have to quantify over branches and time by saying `AG(v == 10)`. If you wanted to express that, in a state machine that controls a pedestrian traffic light, the lights become green for the pedestrians, at some point, might want to say `AF(state == GreenForPedestrians)`: for all possible executions, at some point in the future, the state `GreenForPedestrians` will be reached.

Generally, a temporal property is understood to hold from a particular start state onwards (i.e., for the traces starting from a given state, cf. Fig. 23). So expressing `AF(state == GreenForPedestrians)` means that from some arbitrarily chosen start state, the property becomes true at least once, for all downstream traces. However, if we want to express that this is true for *all* states, and especially for all future ones, we have to wrap this with a second set of qualifiers: `AG(AF(state ==`

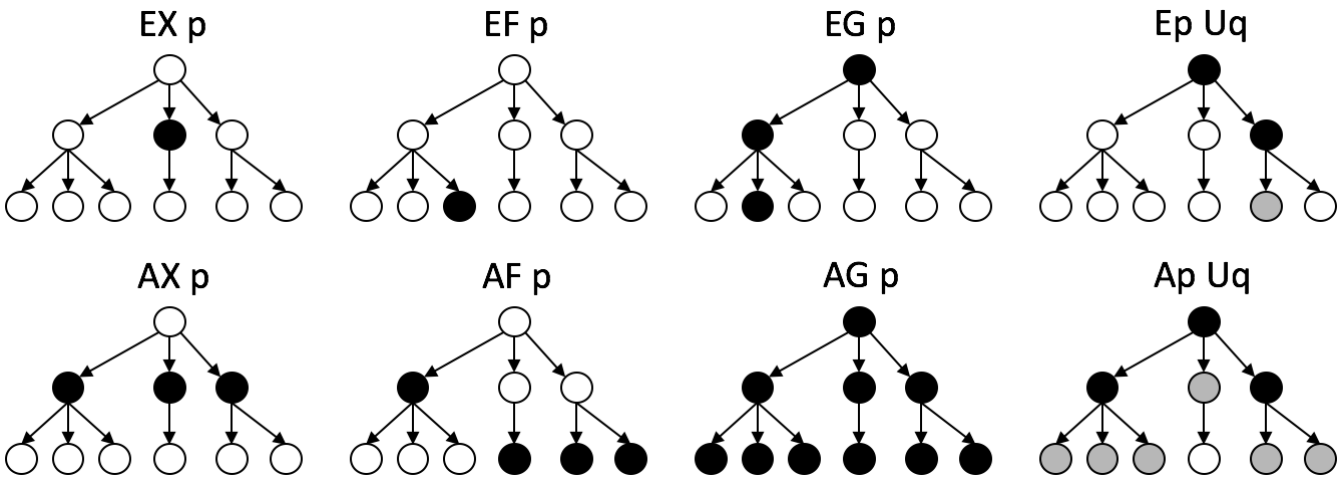

Figure 24. Examples for typical combinations of branch and time operators in the context of temporal logic property specifications for model checking state machines.

`GreenForPedestrians))`. Note that this is not the same as saying `AX(state == GreenForPedestrians)`, because this would mean that for *all* states in all traces, `state == GreenForPedestrians`. Thus, it is generally a good idea to read (nested) temporal formulas inside-out. This style of nested temporal qualifiers is rather typical; we illustrate several of them below.

The properties stated in prose above can be expressed as follows:

1. `AG((state == A) ⇒ AF(state == B))`.
2. `AG((state == A)`
   `⇒ AF(state == B || state == C))`.
3. `EF(state == A ⇒ EF(state == B)`.
4. `EF(state == B) ⇒ (state == A)`.
5. `AG((state == C) ⇒ !EF(state == A))`.
6. `AG(state == A) ⇒ (v > 10)`.

A good introduction to model checking and the various temporal logics can be found in [6].

Typical Patterns While the temporal logic expression can, like all other expressions, be combined arbitrarily, there is a set of typical patterns that occur in the verification of many systems; a catalog of these patterns can be found in [14]. We list some examples that are taken from [42]:

- A device will become *ready* eventually (note that *ready* is a shorthand for any possible Boolean property): `AF(ready)`.
- If we want to express that, globally, for all possible start states, all possible executions will reach the *ready* state eventually, we have to write `AG(AF(ready))`. Because of the outer `AG`, this also means that *all future executions* will eventually reach *ready*. In other words, from the perspective of a particular start state, *ready* will happen infinitely often.
- After something has been *requested*, it will eventually become *available*, for all possible executions: `AG(requested ⇒ AF available)`.
- A specific `end` state will be eventually reached, for all executions: `AF(AG(state == end))`. The difference between `AG(AF p)` (second example) and `AF(AG p)` (this one) is substantial. For example, `AF(AG p)` requires that `p` remains true all the time (`AG`) from some some arbitrary starting point onwards (`AF`). In contrast, `AG(AF p)` requires only that, for all possible executions (`AG`), `p` will become true, eventually (`AF`).
- From every state, it is possible, via one of the branches, to reach the `start` state again, eventually: `AG(EF(state == start))`. Note that this does not mean that *all* executions will be reaching `start` again, so this property cannot be used for induction.
- Some *risky* condition can never happen, for all executions, until some *protection* has occurred: `AG(A(!risky U protection))`.

Boundedness *Bounded* model checking means that the checker considers a limited, i.e., bounded number of steps – you can imagine the checker "simulating" all possible executions of a state machine, but only for the specified number of steps. If a bounded checker reports a property as valid, it means that no property violation has been found within the bounded number of steps it executed; a violation might be found if more steps are considered. So, strictly speaking, a naive use of a bounded model checker cannot prove a property correct, because there is *always* the possibility that the property is violated later. However, by integrating induction into

the overall verification procedure, this limitation can be avoided (cf. the Pacemaker example discussed later).

There are alternatives to bounded model checking, for example, symbolic [26] model checking, which relies on SMT solvers. However, we do not have experience with those and hence do not discuss them any further.

6.3 Model Checking Models

The feasibility of model checking depends on the state space (essentially the size of the branch tree shown in Fig. 23) of the to-be-checked state machine. The bigger it is, the harder it is for model checking to terminate within a reasonable amount of time, and with a reasonable amount of memory. If the to-be-checked system can be modeled as a state machine that only has states, transitions, well-bounded variables (e.g., integers from 0 to 9 or enums) and you can model the occurence of triggering events and the execution of actions as Booleans, you can create rather large state machines and check rather elaborate properties. You can use dedicated model checkers, where you describe your system in the tool's input format (using a transformation, as discussed in Sec. 2.11). Importantly, you represent all interactions with the outside world through (Boolean) events, you don't try to actually describe the behavior of the outside world. The following listing shows the state machine from Fig. 30 encoded for NuSMV:

```
MODULE SM
VAR
  v       : int;
  event   : {none, E, F};
  state   : {start, A, B, B};
ASSIGN
  init(state)   := start;
  init(v)       := 0;
  init(request) := none;
  next(v)       := case
                     state = A & event = E & v < 10 : v + 1;
                   esac;
  next(state)   := case
                     state = start                  : A;
                     state = A & event = E & v < 10 : B;
                     state = A & event = F          : C;
                     state = B & event = E          : C;
                   esac;
SPEC
  ... various properties in CTL.
```

We describe a system based on this approach in [34]; it relied on the NuSMV model checker [9]. Note that the state machine was embedded inside a C program. Properties could be expressed in a DSL based on Dwyer's patterns [14]; the verifier also verified a couple of default properties for state machines (dead states, live transitions). An example is shown in Fig. 25. As mentioned, the NuSMV-based verification only considered the state machine as a black box where interactions with the outside world were modeled as events. While the verification took the (non-)occurence of events into account, it did not consider the effect of those events in the surrounding C program (out events could be mapped

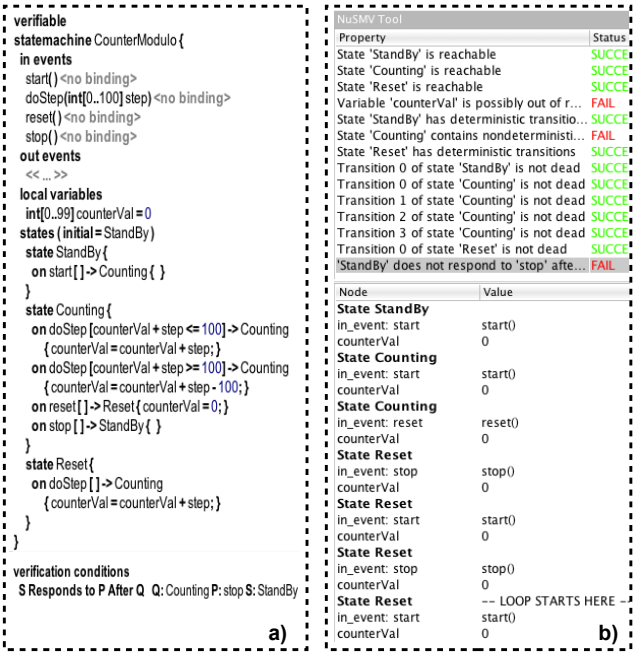

Figure 25. Model checking an example state machine.

to functions). In addition to sending out events, the only other code allowed in actions was setting and reading values of the variables of the state machine. So, even though the state machine was embedded in C, and the actions were expressed with a (severely restricted) subset of C, the machine could be translated to the specific input format of NuSMV. It performed the verification quickly, because the state space was limited. So this is a very good approach if you want to verify the logic of a state machine. However, since the verifier did not take the rest of the C program into account, and because the action code was very limited, this verification was also not terribly useful *to verify a state machine as part of a C program*. We eventually replaced it with full C verification, as described below.

6.4 Model Checking Low-level Code

Functional programs can be verified (largely) with regular first-order logic; we have discussed this in Sec. 5. The reason for this is that they do not have state, so nothing changes over time. In contrast, in imperative languages, the program state (in the form of variables) *does* change over time: this is why those programs must be verified through model checking. It should be obvious that the complexity of such verifications is higher than for functional languages – with the resulting bigger problems for scalability.

Various tools exist for model checking imperative programs. Examples include Java Pathfinder [17] (for Java) or CBMC [21] for C. The Wikipedia page on

Model Checking Tools[38] has a long list of them. We have experience with CBMC, which is why we discuss it in more detail.

Fundamentally, CBMC translates C programs into an internal representation that resembles a transition system . However, from a user's perspective, this internal format is irrelevant – the tool's input is C source code. CBMC also encodes properties in C. More specifically, a major mode of operation of CBMC is to ask it to try to find a way to run a program so that it ends up at a specific C label:

```
1 void doSomething(int a, int b) {
2   ... do stuff ...
3   if ( aCondition(a) ) {
4      somethingWentWrong:
5   }
6 }
```

The `somethingWentWrong` label in C can be used as a target for C's `goto` statement. CBMC (mis)uses them as a marker of a location in a program. So, for example, if you wanted to encode that in a state machine (encoded as a `switch` in C) in state `B` you cannot have a the `v` variable to be greater than 10, you can do this in the following way:

```
1  int v = 0;
2  enum Event { e1, e2, e3 };
3  void theStateMachine(Event trigger) {
4    switch(state) {
5      case A: if (trigger == e1) {
6                  state = B;
7              }
8              break;
9      case B: if (v > 10) { Prop_B_v_violated: }
10             if (trigger == e2) { ... }
11             break;
12     case C: ...
13             break;
14   }
15 }
```

Notice the "property check" in line 9: we use a regular `if` statement, in the context of the particular state, to check if `v` is greater than 10; a label is put inside the body of the `if`. CBMC can be asked to find any possible way to reach this label. Notice that this approach is identical to runtime error checking: instead of the label, you would report an error through some appropriate means.

The property checked above does not really encode any real time dependence; it could be written as `AG(state == B ⇒ v <= 10)` (AG properties are relatively trivial, because they have to hold always, so they can be encoded as just a global check). A more interesting one is, for example, that after `v` has been 10, in the next step, `v` has to be zero (a kind of wrap-around counter). It could be written as `AG(v == 10 ⇒ AX(v == 0))`. This property has to be "implemented procedurally":

[38] `http://en.wikipedia.org/wiki/List_of_model_checking_Tools`

```
1  int v = 0;
2  enum Event { e1, e2, e3 };
3  void theStateMachine(Event trigger) {
4    if (v_prev == 10 && v != 0 ) { Prop_v_10_0_violated: }
5    switch(state) {
6      case A: ...
7      case B: ...
8      case C: ...
9    }
10 }
```

We maintain a variable `v_prev` that contains the previous value of `v`. Once we have this variable, and once we ensure that it is maintained correctly, through more procedural code, we can verify this property with the usual `if` statement.

```
1  int v = 0;
2  int v_prev = 0;
3  enum Event { e1, e2, e3 };
4  void theStateMachine(Event trigger) {
5    // operational code
6    v_prev = v;
7    v++;
8    if (v == 11) v = 0;
9    // property check
10   if (v_prev == 10 && v != 0 ) { Prop_v_10_0_violated: }
11   // state machine
12   switch(state) {
13     case A: ...
14     case B: ...
15     case C: ...
16   }
17 }
```

The good thing about CBMC's approach is that any property check can be realized, simply by "programming" it. The downside, of course, is that the declarative nature of a property specification is completely lost: the procedural specification of a property may be just as error prone as the original implementation code itself. Put in terms of Sec. 2.7, the "goodness" of the specification is lost, it cannot be trivially reviewed. We still retain the redundancy benefit, so the check still has some use.

6.5 Language Extensions and Model Checking

The solution to this dilemma lies in language engineering, more specifically the definition of language extensions that allow the declarative specification of interesting properties, combined with code generation, that "encodes" them in a low-level way for CBMC to check. Assuming the code generator is correct (and it will be, after a while), this solves the problem. In the context of mbeddr, we have developed two sets of C extensions that exploit this idea. We will discuss both of them briefly.

Component Contracts The first example concerns contracts for component interfaces. mbeddr supports components that can be seen as a kind of coarse-grained, statically allocated objects (in the sense of object-orientation). Components can provide (i.e., implement) and require (i.e., make use of) interfaces. Interfaces are similar to Java interfaces in that they define a couple of operations for which the providing component has to

supply an implementation. In addition to the signature, however, interfaces can also specify the semantics of these operations through pre- and postconditions (essentially supporting design-by-contract [27]). The following is an example:

```
interface TrackpointStore {
  void store(Trackpoint* tp)
    pre isEmpty()
    pre tp != null
    post !isEmpty()
    post size() == old(size()) + 1
  Trackpoint* take()
    pre !isEmpty()
    post result != null
    post isEmpty()
    post size() == old(size()) - 1
  Trackpoint* get()
    ...
  query int8 size()
  query boolean isEmpty()
}
```

Preconditions are Boolean conditions that must be ensured by the client when it calls an operation. They typically refer to arguments of the operation as well as on component instance state through (side effect-free) `query` operations. Postconditions express what the implementation of the operation guarantees to be true after the operation has terminated. Postconditions refer to the `result` of the operation, the arguments, the instance state, as well as the instance state *from before the execution of the operation* using the `old` keyword (the latter is a simple form of specifying temporal properties). Alternatively, as the temporal behavior of components becomes more involved, users can also use protocol state machines [37], as the example below shows:

```
interface TrackpointStoreWithProtocol {
  // store goes from the initial state nonEmpty
  void store(Trackpoint* tp)
    protocol init -> nonEmpty
  // get expects the state to be nonEmpty, and remains there
  Trackpoint* get()
    protocol nonEmpty -> nonEmpty
  // take expects to be nonEmpty and then becomes empty
  // if there was one element in it, it remains in
  // nonEmpty otherwise
  Trackpoint* take()
    post(0) result != null
    protocol nonEmpty [size() == 1] -> init(0)
    protocol nonEmpty [size() > 1] -> nonEmpty
  // isEmpty and size have no effect on the protocol state
  query boolean isEmpty()
  query int8 size()
}
```

The key insight here is that the contracts are specified *on the interface*. Code generation then ensures that all components that provide the interface are generated in way so check these contracts. Checks can either be at runtime (through error reporting) or statically (through CBMC). As we have seen above, the code is essentially the same for both cases:

- At the beginning of an operation implementation, use `if` statements to check that each precondition holds; a call to an error reporting function or a label is inserted into the body of the `if`.
- At each return site, check all postconditions, again, with an `if` statement. If the `old` keyword is used, make sure that the old values are stored in a local variable at the beginning of the operation implementation.
- The protocol state machines can also be reduced to code that executes at the beginning and at return sites; in addition, a `state` variable must be maintained in the instance data.

Once a contract has been specified, it can be checked via the directly integrated CBMC model checker, as shown in Fig. 26: the low-level code is generated, CBMC runs on it and produces a low-level result, the mbeddr IDE lifts the result and shows it in meaningful way, i.e., on the abstraction level of the contracts. More details on mbeddr's components and their CBMC-based verification can be found in section 4.4 of [45].

Proving State Machines Correct The second example of CBMC-based verification of C code concerns the correctness proof of a pacemaker as described in [31]. Once again, we have added language abstractions to C to model the behavior of the system (a state machine) as well as the properties: in this case, all properties were Boolean properties depending on the state. Because of these abstractions, specification of the system and the properties could be done on "model level", i.e. at a level of abstraction that is meaningful to the domain. However, because the verification happens on C level, using CBMC, the approach essentially combines the benefits of models (for specification) and code (verification on the level of "the truth").

The paper [31] illustrates two more points worth discussing. First, we also modeled the environment of the to-be-verified system, in this case the heart. The purpose of an environment is to limit the behaviors with which the to-be-verified system has to interact. A well-chosen environment reduces the overall state space, thus making the verification more efficient.[39] For the pacemaker, the environment limits the the frequencies at which a realistic heart might beat, and to which the pacemaker has to react, to 30 - 200 (instead of the full range of integers). The nondeterminism in the heart's selecting that frequency was modeled with CBMC's `nondet` feature. `nondet` is similar to a random value in the sense that any value from a given range can occur; however, CBMC interprets it as "all values can occur", and tries to verify the system for all possibilities. Note that the use of an environment also poses risks, because

[39] In this example, the system could also have been verified without this reduction in complexity; the system was small enough for CBMC.

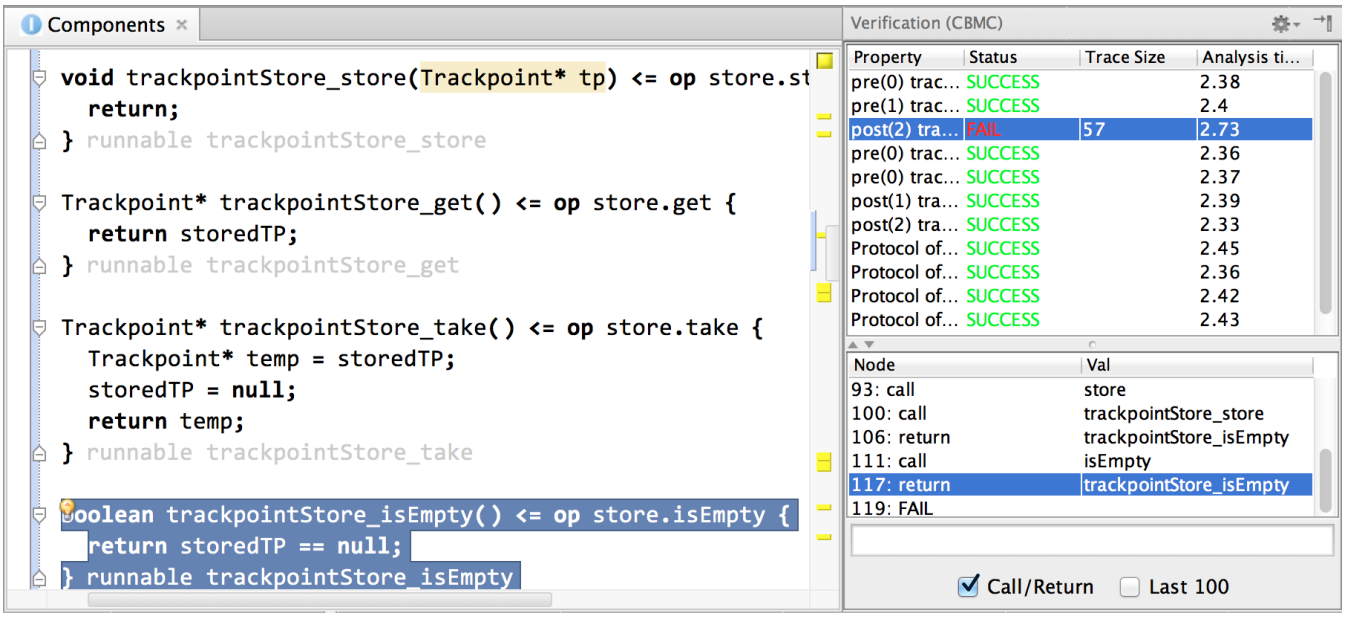

Figure 26. Component verification in mbeddr. The code on the left shows a component implementation for which we check the contract associated with the component's provided interface. The red/green table on the top right shows the set of passed/failed checks. And the bottom right shows the execution trace of a failed check.

no verification is performed for behaviors *outside* of what the environment models. And while a heart really cannot beat with more than, say, 200 bpm, a faulty sensor might very well *report* a higher heart rate to the pacemaker!

The second important point relates to the boundedness of CBMC-based model checking: CBMC does not run the program for an indefinite length of time. This means that is cannot verify that something will never happen in the future. So, to proof safety ("something bad will *never* happen") some additional strategy is needed. In the paper we used induction: we showed that, essentially, the execution of the pacemaker is cyclic, and that the system would be guaranteed to end up in a total state that was previously seen in the trace. This way, we could essentially proof safety "forever". The induction condition and the definition of the total state was also expressed with C extensions in so-called verification harnesses. Fig. 27 shows the ingredients to the approach; the figure is taken from [31] where we describe the details.

Checking Compatibility In component contract example we have shown how to use CBMC, together with C extensions, to check if a (client) program conforms to a contract, optionally specified as a state machine, of a server component. In this subsection we describe how to do this on the level of a model, expressed with a more restricted language,[40] and then using an SMT solver to verify the validity of the state machine.

[40] This was work performed by Alexandra Bugariu as part of the IETS3 project.

```
state machine Pacer {
  in Tick;
  in Sense;
  in Config(lri, vrp);
  out MkPace;
  int c, LRI, VRP;
  state Wait {
    on Sense[c<VRP]->Wait
    on Tick[c>=LRI]->Pace
    on Tick[c<LRI]->Wait{++c;}
    on Sense[c>=VRP]->Wait {
      c=0; }
} }
  state Init {
    on Tick->Wait {c=0;}
    on Config->Init {
      LRI = lri;
      VRP = vrp;
    }
  }
  state Pace {
    entry {send MkPace;}
    on Tick->Wait {c=0;}
  }

environment Heart { nondet smTrigger(Pacer, Sense); }

assign lri: 500 <= lri <= 1500 && lri % 20 == 0;
assign vrp: 0 <= vrp <= 0.2 * vrp;

total state set I for Pacer: smInState(Wait)
  && c == 0 && LRI == lri && VRP == vrp;

inductive for Pacer on Tick
  from: I
  environment: Heart
  conditions: after smIsInState(Wait)
              before smIsInState(Pace)
              exists c == LRI;
```

Figure 27. Pacemaker verification with mbeddr C. In addition to the pacemaker model itself, the code also defines an environment that models the heart as well as the total state of the system and an inductive proof based on the observation that the process is cyclic and the total state repeats.

Consider Fig. 28; the goal of model checking is to see whether a procedural client program conforms to a protocol state machine. In the example in Fig. 28, the

first function is invalid because it tries to write to the file while the protocol is in the `Read` state and because the file `f` is `destroy`ed (because the function returns and `f` goes out of scope) before it is `close`d.

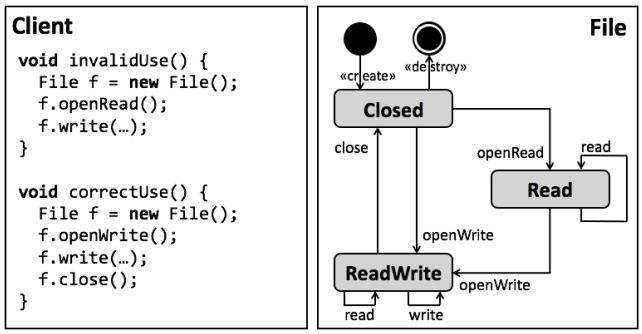

Figure 28. Checking against protocols. We validate the two procedural client programs against the state machine defined by the server.

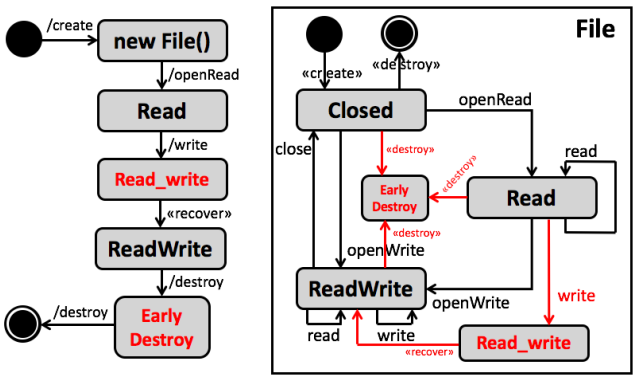

Figure 29. Transformed version of the code in Fig. 28: one of the client programs has been transformed into a state machine as well. And the server state machine has been extended with explicit error states and transitions into these states that occur if an event is unhandled in the original state machine.

The verification relies on the idea of translating the client program into a state machine as well (similar to what CBMC would do internally as well); it is essentially a sequential state machine that uses each program statement as a state. The left part of Fig. 29 shows this.

To verify this, we have to make two decisions. First, how do we model the interaction between the two programs? We considering the method calls (`openWrite`, `write`) as sending events into the `File` state machine (as intuited by the fact that we use the event names as method names). In Fig. 29 we show do this using transition actions (`/action` notation).

Second, we have to define what it means for the client to be invalid: an invalid client sends an event to the `File` at a time when this event cannot be handled, i.e., there is no transition. So we create a modified version of the `File` state machine that has additional states that are entered when (in the original state machine) unhandled events arrive. In the analysis, this is an automated mechanical transformation; in the example in the right part of Fig. 29 we only show the two invalid states that are necessary for the example: if the `File` receives a `write` event while it is in the `Read` state, the machine transitions to the `Read_write` state. We also add an `EarlyDestroy` state into which we transition if we receive `destroy` in any state other than `Closed` where it is handled explicitly. At this level, the verification condition that the model checker would have to verify is: foreach invalid state `IS`, the property `AG(state != IS)` must be verified. This is conceptually similar to the `if` statements with the labels in C, and then asking CBMC to prove that the label can never be reached.

To join the client and `File` state machines, and to see if this client program actually does drive the `File` state machine into an invalid state, we create a merged state machine `M` where the states are the product of the two original state machines. For example, `M` starts with a state `C_newFile__File_Closed` and has a state `C_read__File_openRead` (a valid state) as well as a state `C_write__File_Read_read` (an invalid state). Because this state machine is rather big, we do not show it here; however, the properties for the model checker essentially remain the same: prove that none of the invalid (product) states can ever be reached.

For verification, we implemented a model checker using the Z3 SMT solver; we discuss the mechanics of this in the next subsection below. Overall, this experiment failed, for two reasons. First, we have implemented all the various state machine translation and merging steps with procedural Java code; implementing complex transformations with procedural Java code quickly becomes large and unwieldy. Second, trying to perform model checking with the Z3 SMT solver was tedious and error prone; at this point we are not sure if this is a matter of principle (and we should never have attempted this) or whether we've made low-level Z3 mistakes (for example, we perhaps could have used functions in Z3 to express the temporality of the state variables instead of having thousands of explicit `state[T]` variables).

An alternative would be to map to a model checker like UPAAL [5] directly, so we do not have to encode the evolution of the system over time manually. This needs more investigation. Another alternative approach is to generate C or Java code for the procedural client and the server-side state machine, and then verify this with CBMC or Pathfinder. While this solves all the encoding problems, one has to make sure that the semantics of the model (e.g., number ranges, division semantics, etc.) are reproduced faithfully in C or Java.

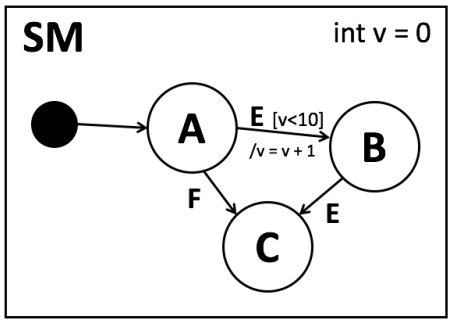

Figure 30. A simple state machine with events used to illustrate model checking with SMT solvers.

6.6 Model Checking with SMT solvers

Total State Transitions As we have discussed before, the propositional logic used by SMT solvers does not encode state evolution over time. This is why we discuss model checking as a separate formalism. However, time can be simulated in an SMT model. Consider a refined version of our introductory state machine, shown in Fig. 30. It can be represented in an SMT solver by expressing a implies relationship of the total state at times `t` and `t+1`, as well as an event `E`: `T[t] && E` ⇒ `T'[t+1]`. This can be read as: if the machine is in some total state `T` at time `t`, and `E` happens, then it will be in total state `T'` at time `t+1`. By total state we mean the current state of the machine (`A, B, C`), as well as the values of all the state machine variables (`v`). For our example, this can be expressed as:

```
enum{A, B, C} state;
enum{E, F} event;
int v;

// this encodes the machine for any step in time t
state[t] == A && event[t] == E && v[t] < 10
   => state[t+1] == B && v[t+1] == v[t]
state[t] == A && event[t] == F
   => state[t+1] == C && v[t+1] == v[t]
state[t] == B && event[t] == E
   => v[t+1] == v[t] + 1 && state[t+1] == C && v[t+1] == v[t]

// this is the initial condition for t == 0
state[0] == A && v[0] == 0
```

Notice how we are *not* constraining the event for time `t+1`. This is because this is what we will task the solver to find out. Remember that an SMT solver solves sets of equations, i.e., it tries to find values for all non-constrained variables so that all equations become true. In this case, the set of unconstrained variables is the sequence of events `event[t_i]` that are processed by the state machine. So, if the solver finds a solution, then it has computed a sequence of events, and thus, implicitly, a sequence of total states.

Encoding Properties To encode an AG property (such as the `AG(state != AnErrorState)` discussed in the previous subsection), one can simply add another constraint:

```
forall t: state[t] != AnErrorState
```

Explicit or Implicit Encoding A naive encoding of the temporal behavior (the variables that depend on `t`) of the state machine would be to replicate all equations `N` times, with explicit indices, where `N` is the number of steps that should be used as the bounds in bounded model checking (determining a good value for `N` can be hard; see the Leakiness paragraph in Sec. 2.13). This leads to `E * N` equations, where `E` is the number of equations, and many more variables. This, however, is not necessarily a problem for modern SMT solvers such as Z3, because they are optimized for dealing with large sets of equations. However, at least in our experience, constructing these equations, i.e., calling the respective APIs on the solver or generating/parsing the input file, can be a performance issue. An alternative encoding might be to use solver-level functions to abstract over `t`, just as we did in the set of equations above. However, if this is possible, is currently unclear to us.

Illustrating the Result As also discussed in Sec. 2.13, one challenge of formal verification in general, and model checking in particular, is to illustrate the result. In our example here, we might want to illustrate the sequence of total states (`state[t], event[t]` and all variables`[t]`) by which the machine ended up in an invalid state. To get this information, one has to trick the solver: remember, that if the solver cannot find a solution, it just stops and reports UNSAT. However, if it finds a solution, it can report it as the set of values for all free variables. So, in order for the solver to tell us which values of the total state lead to an invalid state, the constraint has to reformulated as

```
exists t: state[t] == AnErrorState
```

The resulting model then contains a list of total states for all `t` that lead to an invalid state. The IDE can illustrate this in various ways; what we did in our experiment was to have a table with the values for the total state where the user can click through, highlighting that state in the state machine. We also played with showing the values directly inline in the state machine, thereby animating the machine itself.

7. Wrap Up and Outlook

This booklet has illustrated four techniques for program analysis and verfication: types, abstract interpretation, SMT solving and model checking. These four were selected on the basis of their integratability with DSLs and our own experience.

However, there are many other methods that can be used in the context of DSLs including deductive databases and logic programming [29] (with tools such as Prolog or Datalog), more advanced constraint solvers such as Alloy [28], process algebras [16] and of course theorem proving [33] with tools such as Isabelle/HOL or Coq. Techniques such as discrete event simulation [15] and partial evaluation [19] are also "useless computer science theory" that is becoming more and more use*ful*, especially when combined with DSLs. We have come across some of those, and we have even started to use some of them prototoypically in DSLs. Maybe future versions of this booklet will cover them in more detail.

Acknowledgments

The authors want to thank the following people for their helpful feedback comments: Domenik Jetzen, Krishna Narasimhan, Dennis Klassen, Dave Akehurst, Nora Ludewig, Torsten Goerg and Martin Henschel. The last two gave the most detailed feedback, so particular thanks goes out to Martin and Torsten.

References

[1] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, 2010.

[2] D. Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*. Springer, 2005.

[3] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*. Springer, 2005.

[5] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal—a tool suite for automatic verification of real-time systems. *Hybrid Systems III*, pages 232–243, 1996.

[6] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. A system of patterns: Pattern-oriented software architecture. 1996.

[8] C.-L. Chang and R. C.-T. Lee. *Symbolic logic and mechanical theorem proving*. Academic press, 2014.

[9] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.

[10] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[11] M. Coppo, F. Damiani, and P. Giannini. Refinement types for program analysis. *Static Analysis*, 1996.

[12] K. Czarnecki, U. W. Eisenecker, G. Goos, J. Hartmanis, and J. van Leeuwen. Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, 15, 2000.

[13] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[14] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. IEEE, 1999.

[15] G. Fishman. *Discrete-event simulation: modeling, programming, and analysis*. Springer Science & Business Media, 2013.

[16] W. Fokkink. *Introduction to process algebra*. Springer Science & Business Media, 2013.

[17] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal*

on Software Tools for Technology Transfer (STTT), 2 (4), 2000.

[18] C. A. Hoare. Quicksort. *The Computer Journal*, 5(1), 1962.

[19] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[20] N. Jussien, G. Rochart, and X. Lorca. Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008.

[21] D. Kroening and M. Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014.

[22] K. R. M. Leino. This is boogie 2. *Manuscript KRML*, 178(131), 2008.

[23] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010.

[24] K. R. M. Leino and V. Wüstholz. The dafny integrated development environment. *arXiv preprint arXiv:1404.6602*, 2014.

[25] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. Selim, E. Syriani, and M. Wimmer. Model transformation intents and their properties. *Software & systems modeling*, 15(3), 2016.

[26] K. L. McMillan. Symbolic model checking. In *Symbolic Model Checking*. Springer, 1993.

[27] B. Meyer. Design by Contract: The Eiffel Method. In *TOOLS 1998: 26th Int. Conference on Technology of Object-Oriented Languages and Systems*. IEEE CS, 1998.

[28] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 609–619. IEEE, 2015.

[29] J. Minker. *Foundations of deductive databases and logic programming*. Morgan Kaufmann, 2014.

[30] A. Møller and M. I. Schwartzbach. Static program analysis. http://cs.au.dk/~amoeller/spa/, 2015. Department of Computer Science, Aarhus University.

[31] Z. Molotnikov, M. Völter, and D. Ratiu. Automated domain-specific c verification with mbeddr. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014.

[32] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2015.

[33] B. Pierce and et al. Software foundations. Website, 2016. `https://www.cis.upenn.edu/~bcpierce/sf/current/index.html`.

[34] D. Ratiu, B. Schaetz, M. Voelter, and B. Kolb. Language engineering as an enabler for incrementally defined formal analyses. In *Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*. IEEE Press, 2012.

[35] T. Reps. Program analysis via graph reachability. In *Proceedings of the 1997 International Symposium on Logic Programming*, ILPS '97, Cambridge, MA, USA, 1997. MIT Press. ISBN 0-262-63180-6. URL `http://dl.acm.org/citation.cfm?id=271338.271343`.

[36] T. Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1), Jan. 2000. URL `http://doi.acm.org/10.1145/345099.345137`.

[37] B. Selic, G. Gullekson, J. McGee, and I. Engelberg. Room: An object-oriented methodology for developing real-time systems. In *Computer-Aided Software Engineering, 1992. Proceedings., Fifth International Workshop on*. IEEE, 1992.

[38] T. Szabó, S. Alperovich, S. Erdweg, and M. Voelter. An extensible framework for variable-precision data-flow analyses in mps. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016.

[39] T. Szabó, S. Erdweg, and M. Voelter. IncA: A DSL for the Def. of Incremental Program Analyses. ASE, 2016.

[40] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 2013.

[41] Z. Ujhelyi, G. Bergmann, Ábel Hegedüs, Ákos Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *SCP*, 2015.

[42] F. v. Henke. Einführung in Temporallogik. Website, 2007. `http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/Modellierung.und.Verifikation/SS07/folien01.pdf`.

[43] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1999.

[44] V. Vergu, P. Neron, and E. Visser. *DynSem: A DSL for dynamic semantics specification*, volume 36. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[45] M. Voelter. *Generic tools, specific languages*. Delft University of Technology, 2014.

[46] M. Voelter and B. Pierce. omega tau podcast, episode 243 – formal specification and proof. Website/Podcast, 2017. http://omegataupodcast.net/243-formal-specification-and-proof/.

[47] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 2012.

[48] M. Voelter, A. van Deursen, B. Kolb, and S. Eberle. Using C language extensions for developing embedded

software: A case study. In *OOPSLA 2015*, 2015.

[49] M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with mps. *Software & Systems Modeling*, Jan 2017. URL `http://dx.doi.org/10.1007/s10270-016-0575-4`.

[50] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1999.

[51] A. Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[52] H. Zhu, P. A. Hall, and J. H. May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4), 1997.