Modelling Program Verification Tools for Software Engineers

Sophie Lathouwers
University of Twente
Enschede, The Netherlands
s.a.m.lathouwers@utwente.nl

Vadim Zaytsev
University of Twente
Enschede, The Netherlands
vadim@grammarware.net

ABSTRACT
In software engineering, models are used for many different things. In this paper, we focus on program verification, where we use models to reason about the correctness of systems. There are many different types of program verification techniques which provide different correctness guarantees. We investigate the domain of program verification tools, and present a concise megamodel to distinguish these tools. We also present a data set of almost 400 program verification tools. This data set includes the category of verification tool according to our megamodel, practical information such as input/output format, repository links, and more. The categorisation enables software engineers to find suitable tools, investigate similar alternatives and compare them. We also identify trends for each level in our megamodel based on the categorisation. Our data set, publicly available at https://doi.org/10.4121/20347950, can be used by software engineers to enter the world of program verification and find a verification tool based on their requirements.

CCS CONCEPTS
• General and reference → Surveys and overviews; • Software and its engineering → Formal software verification; • Theory of computation → Logic and verification.

KEYWORDS
Formal Methods; Program Verification; Megamodelling.

1 INTRODUCTION
Program verification (PV) is a field that has always enjoyed very high expectations, and suffered from them as well. Its objectives are mostly to provide ways to prove that a system satisfies certain requirements. The underlying techniques are typically based on rigorous mathematical reasoning or an exhaustive analysis of the state space, thereby giving software engineers stronger guarantees than testing. It is often accepted that to use program verification (or formal methods in general), one needs to specify their system in a formal notation and thus have considerable formal background to do it in a correct and useful way [18].

To simplify, for the rest of the paper we use the established team “program verification” to mean verification (conformance evaluation) of programs (executable models). Hence, it covers generative techniques, testing, model checking, theorem proving, etc, of source code, automata, Petri nets, transition systems, etc.

Adopting verification tools has shown to present not only technical challenges, but also organisational, social and managerial ones [22], similar to challenges faced by advanced model-driven engineering tools [40]. PV tools are particularly difficult, because even demonstrating potential benefits of their use is highly non-trivial and relies on users having very specific knowledge of the underlying techniques. For the tool developers, the tools themselves often serve as a means to an end, as an opportunity to demonstrate the extent of applicability of their techniques, to exemplify the problems that could possibly be tackled, and to enter an existing subdomain. (Some subdomains are accompanied by sets of mature benchmarks [6, 12, 36] which make comparing techniques by comparing tools a very attractive and attainable goal). Thus, many tools stay in a prototype phase, and being actively developed only till a certain point: until the tool can handle the minimal set of benchmarks, or until the deadline for submitting the paper explaining the underlying techniques, or until graduating from a PhD project.

Besides techniques and tools, there are multiple sources of information to consider. Papers themselves are an obvious source, well-archived on publishers’ websites, but requiring high qualifications to be considered readable and understandable. They are also hard-dated, meaning that an average good paper contains detailed comparison of the proposed tool with its existing counterparts, but no comparison or relation to counterparts that were created after the publication. The papers often refer to product or project pages, which are prone not only to being outdated for reasons mentioned above, but also to being removed due to the jobhopping nature of the academic world: when the principal investigator finishes the project and moves to another institution, it is not guaranteed that the project page will be preserved by their original employer. If available, such websites are also wildly varying in the nature of their content: some literally repeat the contents of the papers, while others complement it with valuable information, illustrations, and links.

Another extremely valuable source of information — primarily about the tools and not always about the techniques — is the code repositories. It has become fairly commonplace in recent years to
either release the tools for (limited) public use to enable empirical replicability, or expose the entire development history in a form of versioned codebase (typically through git, occasionally hg or svn). There are at least three benefits of repositories: (1) the artefacts become much more tangible, and only require several natural steps (like cloning the repository) to set themselves up on the user’s computer instead of extracting them from the paper text; (2) the version history is a technically substantiated claim to the amount of work and to its authorship; and (3) linking tools to one another by shared contributors plays the same social role as linking papers by shared coauthors.

To summarise the problems:

- existing techniques are hard to understand and assess their applicability without very deep specific knowledge;
- tools are hard to classify conceptually and appropriately relate to techniques;
- information sources are dispersed, partly unavailable and partly unreliable.

With the vision to open up the arsenal of PV tools and techniques to a broader public of software modellers and, even broader, software engineers, we have developed a megamodel of program verification. The megamodel can help answering questions like “what am I expected to provide as input to use tool X?” or “what other tools exist for the same problem domain as tool X?”, or even “when was the last time the code of tool X was updated?”. We will elaborate on the megamodel and its PV0 – PV6 levels in § 2.

With this megamodel, we show that there are many different types of PV tools, and those types can be grouped in categories that form a hierarchy. Thus, if a tool from one category comes fundamentally short to solve the end user’s problems, it can be considered to seek alternatives in a broader category. To concretise the megamodel, we complete it with a data set into which we have collected information about 420+ PV tools, frameworks and languages, published recently at two top conferences in the PV domain known for their tool paper hospitality: CAV and TACAS. The data set is available publicly on GitHub, with a reader-friendly interconnected hypertextual frontend at https://slebok.github.io/proverb.

We strongly believe that making the data set freely accessible for exploration, makes it an attractive starting point for software engineers to traverse the domain of program verification. § 3 contains more information about the data set, as well as our methods of gathering the data, categorising and enhancing it.

In § 4, we report some preliminary lessons we have learned ourselves by looking at the collected information, level by level, and analysing some of its trends. Further work on enhancing the data set by cross-referencing it with other collections of linked data such as GitHub and DBLP, is ongoing, but the fact that our megamodel splits the PV domain into distinct groups recognisable from prior research, is considered here as a form of evaluation and evidence that the megamodel is viable and useful.

When it comes to related work, we refer to in-depth overviews of problems in adoption of program verification, formal methods, model-driven engineering and domain-specific languages [15, 18, 19, 22, 32, 42, 53, 54, 67, 70, 74]. For the sake of this project, we consider these problems relatively well-known and embrace the fact that most of them are related to tools, their unavailability, low prototypical quality and other concrete problems described above.

Over the years, many ontologies, taxonomies and surveys have been published about program verification. However, these works tend to target either a specific subset of program verification (e.g. run-time verification [29], high-level synthesis [48]) or a specific domain (e.g. vehicular domain [63], smart contracts [37], railway system design [36]). Unlike these works, we do not focus on a specific domain, but aim to deliver both a megamodel to explain tools, as well as an easily accessible (and extensible!) repository with a large data set of classified tools.

The work that is closest to ours is a report [58] that presents several verification techniques in detail. It covers a broad variety of techniques (model checking, verification condition generation and correct-by-construction design). However, they only mention a few tools per technique, whereas we consider all kinds of published tools. Nonetheless, this can also be a nice starting point for engineers.

When it comes to repositories, we also do not claim outright novelty. Over the years, several projects have tried to achieve more or less similar goals. For instance, the Verified Software Repository [14] was intended to become a collection of tools, verified programs, benchmarks and results. Unfortunately, it was last updated in 2009. Schlick et al. [62] have also proposed to set up a repository to make information about formal methods more accessible. They propose a model for a possible repository which includes information such as problem definitions, experiment data as well as tools. However, to the best of our knowledge this repository has not yet been instantiated, and remained a dream. Our work could be used as part of such a repository, if anyone feels inclined to combine their model with our data that matches specifically the “Tool” part in their repository structure. Other directions for future work and consequences of this project, are considered in § 5.

Figure 1: Each ellipse indicates the potential correctness guarantees that can be acquired by using a tool of that level. PV0 tools give the least guarantees of correctness, whereas PV6 tools allow the user to work towards maximum correctness guarantees. Note that these indicate the potential of each level; a tool may only support a little piece of a level.
2 THE MEGAMODEL OF PV-LEVELS

In this section, we introduce the seven levels of our megamodel: PV0, PV1, PV2, PV3, PV4, PV5 and PV6 (see Figure 1). Intuitively, higher levels give the user more correctness guarantees, though typically at the cost of more user effort, and lower level tools are usually less strict and thus do not require as much PV expertise to be effectively applicable.

Since the ultimate goal of PV is to prove correctness of the artefact in some form and within some domain, we will use the classic division of roles in a correctness proof. It was originally introduced by Goldwasser, Micali and Rackoff in 1985 [33], we use the more widespread modern terminology here:

- **Claim**: This is the client's wish or the program's claim, i.e., the statement to be proved.
- **Verifier**: This tool accepts or rejects the claim, but the prover can also be biased and prone to producing false positive arguments. The verifier has some way of checking the arguments and, depending on its verdict, declaring the claim accepted or rejected. We will be illustrating each of the PV tool levels with explanations, examples and also differences on this simple scheme. In the subsequent diagrams we will also use green colour to highlight the main contributing elements that make someone decide to use a tool of this particular level.
- **Prover**: This tool plays a role of a verifier, and a prover does not exist since the verifier does not need any arguments on top of the ability to observe the given model.
- **Solver**: This tool can check that all these properties indeed hold. The more complex the model, the more difficult it could be to make such a solver for it: for example, uniqueness is relatively easy to check on strings (which we assume for transition labels in our previous example), but it is noticeably harder to define and enforce even on database records, where single columns (such as "first name") often contain non-unique values, and combinations are often unreliable due to incompleteness and subtle tolerable inconsistencies (such as a phone number mismatch). From the correctness perspective, a PV1 tool makes a model of their wishes, often taking a form of an almost-consistent artefact with holes to be filled, then one can build a tool to fill in those gaps and infer them from the context. Sources of information can be different, ranging from domain common sense (for example, we obviously want our parallel programs to not get stuck waiting for one another's resources) to constraints and instructions explicitly specified by the user. In a sense, if we want to consider the Eclipse Modelling Framework as a proof system, it would fall into this category because it can produce the textual code of classes that conform to the inheritance structure and the interfaces specified in the class model. In PV, such programs are often said to solve problems of synthesis and repair, and use generative techniques to create test data, repair known categories of defects, implement queries, generate neural networks fitting for a particular grid, or just to configure a universal algorithm with automatically obtained balanced values. PV2 tools help users to

**[PV0]** Software engineers always work with abstractions and models of reality. Once a software entity satisfies the three properties of the modelling theory [65], it can be seen as a model. These three are the mapping property (elements of the model represent some elements of the real entity being modelled), the reduction property (only the most important aspects of the real entity are being modelled and others are being abstracted from), and the pragmatic property (the model has a purpose). **Formal models** are a subset of such models, which are clean and well-formed, and often built with the use of some existing mathematical theories. For example, formal models often cover domain-specific variations of automata. In PV0, such a formal model may exist but it is often implicit and is used neither to obtain nor to verify any correctness guarantees.

**[PV1]** Once a formal model can be operated on by a software system, it can also be automatically checked for internal consistency and well-formedness, by a **model solver**. For instance, if the underlying theory states that a model of a process is some specific automaton with one starting state and one or more final states, and all transitions labelled with unique names, then a solver can check that all these properties indeed hold. The more complex the model, the more difficult it could be to make such a solver for it: for example, uniqueness is relatively easy to check on strings (which we assume for transition labels in our previous example), but it is noticeably harder to define and enforce even on database records, where single columns (such as "first name") often contain non-unique values, and combinations are often unreliable due to incompleteness and subtle tolerable inconsistencies (such as a phone number mismatch). From the correctness perspective, a PV1 tool

**[PV2]** The opposite situation is also commonplace: if a user makes a model of their wishes, often taking a form of an almost-consistent artefact with holes to be filled, then one can build a tool to fill in those gaps and infer them from the context. Sources of information can be different, ranging from domain common sense (for example, we obviously want our parallel programs to not get stuck waiting for one another’s resources) to constraints and instructions explicitly specified by the user. In a sense, if we want to consider the Eclipse Modelling Framework as a proof system, it would fall into this category because it can produce the textual code of classes that conform to the inheritance structure and the interfaces specified in the class model. In PV, such programs are often said to solve problems of synthesis and repair, and use generative techniques to create test data, repair known categories of defects, implement queries, generate neural networks fitting for a particular grid, or just to configure a universal algorithm with automatically obtained balanced values. PV2 tools help users to
create software artefacts — either by generating them from scratch or by providing significant assistance in the incremental process of creating them semi-automatically. If the output of a PV2 tool is expected to be processed by another automated component, then the tool belongs on a higher PV level.

**[PV3]** Combining the two components into a symmetric setup (cf. Figure 1), in the simplest case we get a situation when a user explicitly states what properties of a formal model they wish to have (beyond well-formedness), and there is an automated property checker, conceptually decomposable into two parts: a prover that turns each property into a convincing argument and a verifier which validates the convincing power of such arguments. A typical example of a property checker allows the developers to add assertions to their code, specifying preconditions, postconditions and invariants around a code fragment, thus allowing additional formal ways to validate its correctness. These assertions do not have to be deployed to the end-user, but serve as a powerful tool for the developer to verify the product beforehand. Some checkers have a very extensive formal language to write properties in, usually a variant of some special kind of logic (e.g., temporal logic [45, 57]).

**[PV4]** On the previous level, the burden to create verifiable properties, was on the end-user of a tool: assertions had to be explicitly written, and invariants had to be provided. However, in some cases it is possible to automate the creation of desired properties as well as their verification — since such techniques require an extensive specification of the desired behaviour, and often focus on only one paradigm, we call them monoverifiers. They are very useful tools in debugging, because if used correctly, they can significantly lower the chances of having a particular category of defects, sometimes up to eliminating the very possibility of such a defect ever occurring. For instance, think of a parallel system being checked for deadlocks or a garbage collector checked for the lack of memory leaks. Essentially, monoverifiers verify that the supplied formal model corresponds to the expectations of their own built-in specification.

Some monoverifiers offer a choice of checking one or more of a larger set of correctness specifications, in which case we still classify them to belong to PV4, even though the mono- prefix no longer fits — as long as the end-user has no direct control over the specifications themselves. Also most monoverifiers embrace the fact that their generated properties cannot always cover the end-user’s needs, and allow for direct manual specification of additional properties — which allows us to claim that PV4 functionalities are a strict superset of PV3 functionalities.

**[PV5]** When the tool users have an opportunity or obligation not only to specify which properties of the system to check for or how to infer them, but also to build their own specifications, we get to specification compilers. Such compilers usually have a language used to write specifications in, sometimes based on a domain-specific notation, and support this entire language by compiling its instances in some automated way to verify their correctness and compatibility. With those, you can build your own specifications of memory management strategies, your own communication protocols, and so forth. To continue with examples from the previous paragraph: when a PV4 tool could check for deadlock freedom, a PV5 tool would require a formal specification of the concurrency model, accompanied with a definition of what constitutes a deadlock state. Obviously, some PV4 tools are built on top of PV5 frameworks by essentially supplying a useful singular model.

**[PV6]** If your program verification tool can not only handle different specifications, but also infer correctness of the proof of the needed property, then it belongs among the proof assistants. This category is the most powerful one that we have encountered, which means both that it is the hardest and most demanding to use, as well as capable of producing the strongest guarantees. However, as one can see from the diagram we provided, it bears some similarities with the PV1 level, since there is very limited automation and tool support in composing the arguments for correctness. The proof needs to be written by the end-user, and the tool can only offer some assistance in verifying that the proof is indeed correct. Some of PV6 tools can feel to their users as if they also help them to compose the steps of the proof, but under closer inspection this help comes from the tool knowing which proof step would succeed in
reaching a user-stated goal, and not from the tool relying on some generative algorithms. Within the claim/prover/verifier paradigm, proof assistants offer powerful techniques on behalf of the verifier and not the other two components.

3 DATA SET OF VERIFICATION TOOLS
To help users find a suitable PV tool, we have prepared a data set of tools categorised according to the megamodel we have just explained in § 2. The data set, called ProVerB, is available at\(^1\): [https://slebok.github.io/proverb/](https://slebok.github.io/proverb/).

This section explains how the data set has been created (§ 3.1) and presents some statistics about the data that was gathered (§ 3.2).

3.1 Methodology
A methodological overview can be seen in Figure 2. Below we will describe each step of our research method in detail.

3.1.1 Choose data sources. To find PV tools to include in the data set, we looked into two popular conferences about verification of systems. Namely, the International Conference on Computer Aided Verification (CAV) and the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). In the CORE conference ranking they are classified as having respective ranks of A* and A. We have decided to use papers from these conferences since both accept and welcome papers about tools. Therefore, we could expect a relatively high percentage of tool papers. Moreover, CAV and TACAS started with artefact evaluations in 2015 and 2018 respectively, therefore we expect that many of the tools presented here will also be available. We have looked at tool papers from TACAS 2016–2021 and all papers from CAV 2017–2021. We chose to use these recent years as this makes it more probable that tools are still findable and possibly maintained, yet still limited ourselves to the last 5 years of each conference in order to gather a substantial amount of data.

This step resulted in 460 papers: 347 from CAV and 113 from TACAS.

3.1.2 Identify tools in papers. Next, we read all the papers from the selected conferences with the intent of identifying tools that are presented in each paper. For each paper we checked whether it contained a reference to a tool. If so, then we would tag this as one of the following claims:

- **Presents**: the paper introduces a new tool;
- **Extends**: an existing tool gains new functionality in the paper;
- **Expands**: the paper uses an existing tool as a basis for building another tool;
- **Uses**: the paper uses an existing tool for a case study or to check the correctness of an approach.

We only included tools that provided some form of correctness guarantees, to avoid including too many entries in ProVerB. This still left us with some entries that were later reclassified as not belonging to the PV domain (usually from misinterpretations of claims “we use library X”).

3.1.3 Collect data. As a format for storing entries in the data set, we have chosen Markdown. This provided the lowest entrance barrier and maintenance cost, still combined with the opportunity to add structure to the data (in our case, in the form of # ###-level sections). By choosing this format, we also hope to make it easy for other people to contribute to the data set in the future, since GitHub, our hosting platform for the data, even provides inline editing functionality for Markdown pages.

After some pilot classifications we set up a generic template for tool pages, which has proven to be quite resilient, and after the first couple of sprints it stayed stable and unchanged till the current moment. This template included a section for all the information that we were interested in for a tool, namely:

- Name
- Domain or application field
- Self-identified type of the tool
- Input that is required from the user
- Input format
- Output that is produced
- Internal working of the tools, such as which tools it uses as a backend
- External relations to other tools, such as those that were compared to this tool in the paper
- Links to project pages, repositories and related papers
- Dates when the tool and its documentation were last updated

Next, we created a page based on this template for each tool that was identified in the previous step, by reading a corresponding paper and extracting the information. We also collected information from the code base and project website if these were available. Some sections were left empty if the data was not available (e.g., the last modification data for tools without a repository). If at least two tools referred to another tool, e.g. because it was used as a back end or as a framework, then this tool was added as well and received its own entry.

Some tools that we encountered were developed as prototypes, up to the point that these did not have a name at all, nor a link to an implementation. We decided to exclude such tools since it is difficult to find information about a tool without a name. However, some tools included an artefact, which was mostly still reliably available, so we included this link in the entry whenever it existed.

Aside from the information mentioned above, we have also started adding tags as textual annotations. Tags are used to indicate whether a tool targets a specific language, domain, technique, etc. This should make it easier for users to find suitable tools. For

---

\(^1^\)The data set has also been archived at [https://doi.org/10.4121/20347950](https://doi.org/10.4121/20347950)
example, there is a tag for tools that target C programs, a tag for neural networks, a tag for hardware verification, and so forth.

The data set also includes pages about several specification formats. A page for a format was created if the format was not tool-specific, if it was used by more tools than one, or if it was for some other reason conceptually separate from the tool.

3.1.4 Define megamodel. When the tool pages had been written, we contemplated the initial setup of the megamodel based on similarities between tools. The first version was already based on the input that the user has to provide, ranging from the no input at all (besides the already existing software artefact), to assertions, properties, specifications, theories, and proofs. Several refinement iterations later, based also on consulting the already available domain knowledge [33, 72], we have arrived at the version presented previously in § 2.

3.1.5 Classify tools. Having designed the initial megamodel, we started the process of classifying all the tools. Based on the tools’ semi-structured description (cf. § 3.1.3), we have assigned each to a PV level. While doing that, we have also consistently provided a short description motivating this classification by explaining what the tool does. In that way, a tool with a description “verifies properties of a user-specified memory model” was clearly placeable at PV5, and the one with “checks user-specified properties and memory-safety of C programs” was easily marked as deserving PV4. To prevent misclassifications, the authors actively double checked each other’s verdicts and had extensive discussions about arguable conclusions.

Aside from the PV0–PV6 levels, there are two other categories: “no PV” for false positives and “frameworks” for a possible level mixture. We used “no PV” to explicitly exclude entries that ended up, after close consideration, not performing any PV-related tasks. Such entries were mostly about specification formats, but also about IDE plugins, unrelated programming languages, libraries not performing any PV tasks, etc. We felt that something like an alternative user front end or a linear programming library do not belong to PV0 either. “Frameworks” were used to classify collections of tools: in many cases it was possible to determine the primary objective of the collection and assign a framework to a proper level as well, but in other cases such an assignment has not been deemed sensible. For example, “Alloy” is used to refer to the Alloy Analyzer (which has its own entry on PV3), or to the input level of the Alloy Analyzer, or to the entire ecosystem of Alloy models and their verifiers, — and is not consistently PV-classifiable without disambiguation.

3.1.6 Identify trends. Finally, after we classified all the tools, we could start to identify trends in each level of the megamodel. These trends could be identified based on the short descriptions that were written in the previous step, and require only occasional lightweight double checking with the full data entry or the text of the underlying paper. We will discuss these trends in more detail in § 4.

3.2 Data set statistics

The data set contains 384 tools, 25 specification formats and 66 tags. The tools are split over the PV levels as follows:

- PV0: 16 — cf. § 4.1
- PV1: 88 — cf. § 4.2
- PV2: 82 — cf. § 4.3
- PV3: 68 — cf. § 4.4
- PV4: 99 — cf. § 4.5
- PV5: 14 — cf. § 4.6
- PV6: 13 — cf. § 4.7
- No PV: 44

Table 1 gives an overview of how many tools were identified in the CAV and TACAS proceedings.

<table>
<thead>
<tr>
<th></th>
<th>Tools</th>
<th>Prototypes</th>
<th>No tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAV</td>
<td>228 (49%)</td>
<td>36 (8%)</td>
<td>89 (19%)</td>
</tr>
<tr>
<td>TACAS</td>
<td>13 (2%)</td>
<td>0 (0%)</td>
<td>19 (4%)</td>
</tr>
<tr>
<td>Overall</td>
<td>322 (69%)</td>
<td>36 (8%)</td>
<td>108 (23%)</td>
</tr>
</tbody>
</table>

Table 1: An overview of how many tools were identified in the CAV and TACAS proceedings.

4 TRENDS IN PV LEVELS

In this section we identify different subgroups within each PV level of the megamodel.

4.1 PV0: Potential tools

At the time of writing of this paper, ProVerB had 16 tools on PV0. 13 of them provide facilities to work with various kinds of seemingly formal artefacts: grammars, regular expressions, automata, decision diagrams, session types, and floating point numbers. However, there is simply not enough formal rigour in the way these tools operate these artefacts, for us to consider them truly a part of the program verifier’s arsenal. As an example, consider ANTLR [55]: given a grammar, it generates a parser for it. However, it does so without the grammar being perceived, modelled and transformed as a mathematical object. If the user provides ANTLR with a grammar which is unconnected or ambiguous, then the generated parser will be faulty.

Two remaining PV0 tools are, in fact, repositories: Ceramicist [34] and Prosa [35] are libraries that store formal artefacts (definitions and proofs) but by themselves neither provide arguments about their correctness, nor verify those (both rely on Coq). The last PV0 tool is Smt-Switch [47], a collection of abstract classes that, if inherited from and implemented, can help integrate SMT solvers—again, this library by itself definitely is related to the PV domain, but does not help bring any correctness guarantees.
What all PV0 tools have in common is their position on the verification diagram we have shown in § 2: they are claims without arguments, without a prover and without a verifier. The claims can be formal, but the surrounding context does not qualify as PV tool support.

4.2 PV1: Essential tools
Out of 88 tools on the PV1 level, 18 can be seen as frameworks enabling their end users to work with certain models/abstractions in a formal way. For instance, Frama-C [20] contains functionality to treat C programs as formal artefacts and thus can be used to build different program analyses on top of it; BINSEC [24] provides similar functionality and opportunities to implement binary level code analysis; there are comparable tools that deal with Büchi automata, symbolic automata, decision diagrams, temporal logic formulae, etc. 7 more tools could be seen as limited frameworks that are developed specifically to compare two models in a formal way. For example, SPAN [9] computes whether two protocols are indistinguishable, and RABIT [1] checks inclusion of languages generated by two Büchi automata. Another 6 tools can be seen as normalisers that bring a given model to some well-defined canonical state: Mealy machines and Büchi automata can be automatically minimised, quantified Boolean formulae can be simplified and turned into dependency quantified Boolean formulae, etc.

21 different PV1 tools are linters, type checkers and checkers of other kinds of properties that are fixed and hardcoded into the tool (we will see checkers of user-specified properties on PV3). Such properties can include conformance, semantic preservation, type safety, automata emptiness, safety of Markov decision processes, thread safety, etc. Reachability and termination analyses, due to their internal workings, we count towards another category, which includes metric calculators and tools that compute a set of possible states of a model or infer ranking functions, or compute upper and lower bounds of something — there are 30 of them in total.

Finally, the remaining 6 tools can execute models, simulate their behaviour, (partly) visualise them and resolve them otherwise: CabPy [7] solves a two-player reachability game, Oink [69] solves a parity game, jcostress [64] and PROVER [61] execute test cases in a specific order, CLEAR [8] and dtControl [5] visualise the problematic part of a labelled transition system and previously externally synthesised controller code, respectively.

4.3 PV2: Creational tools
There are 82 tools in PV2. The largest identifiable group, with 37 members, consists of tools providing correct-by-construction artefacts given a specification: some synthesise a controller from an LTL formula, others generate a dynamic neural network for a given grid, some generate tests for a given circuit, while others specifically generate classes that attempt to violate given properties. This group of tools can produce fairly formal artefacts that are automatically verifiable, but they do not provide any verifier means themselves. 8 more tools perform limited versions of the same process, generating only enough content to fill in holes in an already partially existing model or program. For instance, r-DIGITS [26] fills holes in a given loop-free program from a probabilistic specification of its desired behaviour, and MOVEC [21] performs aspect weaving. Two more tools (DIGITS [2] and TarTar [43]) specifically propose repairs as code fragments meant to substitute existing code fragments assumed to be faulty.

The second popular group contains 19 tools that encode or transform the artefact from one format or formalism to another. This group covers tools for sequentialising parallel C code (MU-Seq [66]), or transforming irreversible programs into reversible circuits (ReVerC [3]). There are several tools on this level that operate on temporal logic formulae, making a timed automaton (MightyL [17]) or an Electrum model (Cervino [56]) or another temporal logic formula in a different dialect (MLTLConverter [46]) from them.

7 tools can be used to refine specifications: for instance, by inferring type annotations from an untyped program such as Typetec [38], or generating permission pre- and postconditions for Viper programs such as Sample [25] does.

Finally, 9 tools generate configurations or settings for other tools, such as PeSCo [59] which generates the best fitting configuration for CPAchecker [13] that fits previous experiences; or SATzilla [75] that decides which solver to call per instance based on predictors.

4.4 PV3: Property checking tools
PV3 currently has 68 tools. Within PV3 we can clearly identify three main subgroups: property checkers, assertion checkers and program repair tools.

The first group consists of 36 tools that check properties for some form of model such as automata or network models. For instance, STAMINA [51] can be used to check properties of infinite-state continuous-time Markov chains.

The second group consists of 27 tools that check assertions for concrete artefacts. For example, SecC [27] can check information flow properties, expressed as assertions, for C programs.

Three tools: Forester [39], SymDIVINE [50] and VeryMax [16] – fall in between these two groups. The first two of these work on LTL formulae as properties, but apply them on real C/C++ code (SymDIVINE allows both “normal” assertions and LTL formulae). VeryMax works both on programs (C/C++) and models (transition systems).

Finally, there is also a small group of tools that focuses on program repair. AllRepair [60] and NNRepair [68]. These tools both identify faults in the program, like other tools in PV3, and they also propose a way to fix it.

4.5 PV4: Specification checking tools
Currently, PV4 is the largest category with 98 tools. The largest group (50) of tools within PV4 are the solvers. These tools produce a satisfiability result for SAT (satisfiability), SMT (satisfiability modulo theories), QBF (quantified Boolean formulae) or CHC (constrained Horn clauses) problems. Because these tools verify a specific property (namely, satisfiability), one may have expected to find them in PV1. However, these tools typically generate interpretations for the given problem to show that it is (un)satisfiable. So, internally each of these tools consists of two essential parts: the property generator which generates the interpretation and the verifier which checks whether this interpretation makes the formula satisfiable. This group also contains the tool that is referred
to the most often in our data set — namely, Z3 [23]. It belongs to SMT solvers together with 17 other tools; there are also 22 SAT solvers; 3 CHC solvers; and 6 solvers of other kinds.

Many PV tools from other levels encode their problems into satisfiability problems and then use one of the tools in this group as a back end.

PV4 also includes 23 tools that generate properties or check built-in specifications typically depending on the domain that the tool targets. Some examples of built-in specifications that are checked, include memory safety, data-race freedom, termination and absence of runtime errors. Many of these tools also provide support to check user-written properties. For instance, Gobra [73] can check user-written assertions for Go programs as well as memory safety, data-race freedom and crash safety.

Finally, there is a small group of what we can call language workbenches [31], and we strongly suspect that there are more of this kind that escaped our selection only because nobody published about them directly at CAV and TACAS recently. A language workbench was envisioned in 2005 as a set of tools aiding the language engineer to design, implement and integrate a collection of domain-specific languages into one unified solution. Some of the popular language workbenches in model-driven software engineering include Xtext, MPS, MetaEdit+, Rascal and Spoofax. The two language workbenches that we have found mentioned for the domain of program verification, were DLC [28], which can automatically generate distributed implementation of concurrent systems modelled in the LNT language, which can be verified using the CADP toolbox; and PrDK [41], a development kit for programming communication protocols.

### 4.6 PV5: Fully controlled verification tools

Continuing the same trend, on PV5 we see a uniform group of 14 verification workbenches. These are tools that allow users to write their own specifications and combine these together with desired properties into a formal mathematical representation. These formal representations can then be compared with representations of programs or their properties for the verification. Users can have very fine-grained influence on the results of these tools because they are allowed to write their own specification. For example, Attestor [4] allows the user to specify the initial heap configuration and the behaviour of the garbage collector that should be taken into account when verifying a property for a Java program. Similarly, UP-PAAL [10] is a workbench for automatic verification of safety and bounded liveness properties of real-time systems modelled as networks of timed automata.

### 4.7 PV6: Proving tools

All 13 tools in PV6 are proof workbenches. Many of the tools in previous categories give a yes/no answer to indicate whether a property holds, and in any case allowing at most some influence on the property generating and handling process, but not on the final proof. PV6 tools, however, will help the user to construct and infer the correctness of a proof that shows why a property is true or false. Some well known tools in PV6 are Coq [11], Isabelle/HOL [52], Lean [49] and Vampire [44]. Their comparison is a highly nontrivial task even for professional mathematicians [71].

### 5 CONCLUSION & FUTURE WORK

Our contribution of this paper is two-sided. On one side, we have analysed a fairly complex domain and turned one of the commonly used visualisation of its core processes into a full fledged megamodel that helped us to split the domain into much more intelligible smaller categories. On the other side, we have processed hundreds of academic papers published across several recent years, classified them according to the proposed megamodel and generated a user-friendly website allowing software engineers to compare and assess papers in a bit more secure, complete and safe way than before.

The megamodel that we have presented, identifies the different type of program verification tools that we found existing or that can possibly be made to exist. This megamodel is based on the classic division of roles in a correctness proof as introduced by Goldwasser et al. [33] that is currently accepted by the computational community. Our megamodel divides the different types of tools into seven categories: PV0, PV1, PV2, PV3, PV4, PV5 and PV6. These categories are increasingly more demanding and increasingly more powerful: it is possible to gain some benefit from a PV0 or PV1 tool within the first day of being introduced to it, but much further refinement and improvement might not be possible; on the other side of the spectrum, PV6 tools can do almost everything, and require a relevant PhD degree to operate. Thus, there is no discussion on “what is the best PV level”, just a classification that helps to match a tool to customer needs.

To bring the megamodel to life, we have designed a metamodel to hold semi-structured information about a PV tool, including its PV-level, name, input/output, etc, and instantiated it for all tools that we have found being mentioned and used in the last five years of two top conferences in the field: CAV and TACAS. Our data set at the time of writing consists of 420+ tools, formats and libraries. By setting up a megamodel as well as a dataset, we hope to provide both a theoretical as well as a practical starting point to get into the world of PV tools and methods. A good starting point for browsing and exploring ProVerB would be its hypertext frontend: [http://slebok.github.io/proverb/](http://slebok.github.io/proverb/) which also contains links to other sites (GitHub, DOIs, etc) for each tool.

One of the most important things to consider for the future is how to keep the data set up to date. To achieve this, we think that it is essential that other users can easily contribute to the project, e.g. through pull requests. It would also be nice to further extend the data set, e.g. by including other conferences. We are actively looking into APIs of related linked data repositories to extend the data set and automatically update information such as the timestamp of the last commit.

#### 5.1 Threats to validity

**Conclusion validity.** All the PV-classifications that we have performed, come from our personal interpretations of the contents of a fairly large body of fairly complex academic papers. Thus, it is possible that some tools have been misclassified as belonging to one level while they actually belong to another level. To prevent misclassifications, the authors were actively double checking each other’s verdicts and had extensive discussions about arguable conclusions. Eventually we plan to reach out to authors of all tools included in ProVerB individually, with a detailed explanation of the
seven PV levels and a request to review our summaries and refine them, possibly leading to reclassifications. If such a community effort causes a noticeable resonance, it would be possible to eliminate this threat entirely.

**Internal validity.** Since our project is more of an observational and classificational nature, and we do not attempt to establish any causal relationships, internal validity is not among our major concerns. Once we start enhancing the data set with other sources of linked data (such as GitHub and SpringerLink API), it might become more relevant to correctly establish contributor identity equivalence across multiple platforms with varying usernames and non-stringly matching names.

**Construct validity.** As we have explained in § 2, our megamodel was designed based on the classic division of roles in a correctness establishing setup as described by Goldwasser, Miccali and Rackoff [33], in the modern reinterpretation by Wigderson [72]. By reusing a model that originates from the right domain, we hope to have found a mature foundation that will allow us to classify any possible tool in the future by matching its components and concepts to the claim, the prover, the arguments and the verifier. Only if we encounter future tools that do not fit into this model, will we have to redesign the megamodel again. However, in the works of Wigderson [72], generalising the notion of a proof from being a unidirectional communication from the prover towards the verifier, to a bidirectional series of communications, handling interactivity, errors, randomness and other natural aspects of computation, has opened a lot of doors and led to the discovery of a number of complexity classes with a distinctly higher expressive power. For instance, relying on more than two verifiers at the same time is not uncommon in PV, but this is mostly done for practical considerations such as trying all available ones to watch only the fastest complete its proof. It is neither considered nor suspected that multi-prover or multi-oracle PV tools can lead us to a broader computational class. Since this has not been researched or established before, we also do not consider such multi-tier setups as one of the PV-levels explicitly.

**External validity.** We have gathered data from publications at CAV and TACAS, which seemed like a good choice of information since both favour papers about program verification tools to non-tool papers and non-PV content. However, there are more venues that target the program verification field (POPL, PLDI, FASE, LICS, etc.). It is unknown at the moment what biases we have created in the data set by limiting ourselves to only CAV and TACAS and related papers, techniques and tools. While limited, the number of tools (380+) included is significant, and they seem reasonably spread out among the different PV levels. We see that the most popular tools are included, partly because we also include tools if they are referred to by at least two other tools. So, while perhaps limited, we think that this is a good starting point for the data set.

**REFERENCES**


Modelling Program Verification Tools for Software Engineers MODELS ’22, October 23–28, 2022, Montreal, QC, Canada


