A Heuristic for the Automatic Generation of Ranking Functions*

Dennis Dams‡ Rob Gerth§ Orna Grumberg¶

Abstract

The duality between *invariance* and *progress* is fundamental in proof techniques for the verification of programs. Proving invariance requires the construction of invariants, while progress proofs hinge on the identification of appropriate ranking functions. With the recent interest in automated verification techniques, the topic of automatic generation of invariants is facing a revival of interest.

In [14] it has been shown that temporal properties of reactive systems can be proven via finitary abstractions if those abstractions comprise a notion of acceptance conditions, like ω-automata. Based on this, that paper concludes that there is a strong need for devising effective heuristics for generating such conditions. In this note, we address this issue. We suggest a simple heuristic in the spirit of, and combining well with, the popular *predicate abstraction* approach to the automatic generation and refinement of invariants. The presentation is non-technical and guided by examples.

1 Introduction

Model checking is a useful technique to automatically determine whether a finite-state program satisfies a temporal logic specification. The technique is often quite efficient in time but suffers from high space requirements that limit its applicability. One of the most useful approaches to reducing the space requirements is *abstraction*. An abstract model is smaller in size (number of states and transitions) but typically contains more behaviours than the concrete model. It is therefore suitable for verifying program properties that are universal in nature, i.e., properties that refer to *all* behaviours of the program. For every universal property (written in logics such as ACTL*, ACTL or LTL), if it is true of the abstract model then it is guaranteed to be true of the concrete model. If, however, the property is false in the abstract model then nothing can be said about its truth in the concrete model.

Both safety (invariance) and liveness (eventuality) properties are expressible in ACTL* and therefore can be verified using abstraction. Unfortunately, liveness properties often do not hold in the abstract model even when they are true of the original program. This is because the abstract model may contain spurious cycles, i.e. cycles that have no corresponding ones in the concrete model, along which a desired eventuality is never reached. An abstract model

*The ideas presented in this note have been developed during a visit of the first two authors to the Technion in Haifa, Israel, hosted by the third author, during the fall of 1995. We thank the Netherlands Organisation for International Co-operation in Higher Education (Nuffic) for its financial support.

‡Dept. of Electrical Engineering & Dept. of Math. & Comp. Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven. E-mail: `d.dams@tue.nl`.

§Intel Architecture Business Group, SCL, 5200 NE Elam Young Parkway, JFT-104, Hillsboro, OR 97124-6497. E-mail: `robgerth@ichips.intel.com`

¶Computer Science Dept., Technion, Haifa, Israel. E-mail: `orna@cs.technion.ac.il`

containing such cycles will not satisfy the eventuality property, thus, we will be unable to derive its correctness for the concrete model.

A common way to remove additional cycles is to eliminate abstract transitions by partitioning abstract states ([15, 5]). However, partitioning abstract states alone is not enough in general. More precisely, given a system and an LTL property which is satisfied by it, in order to demonstrate this via a *finite* abstraction of the system, it is necessary to consider abstract transition system with acceptance conditions — this follows from the results of [14].

In this paper we suggest an approach that is complementary to the partitioning of abstract states, based on the classical well-founded method used to prove liveness properties (e.g. termination) for general programs. A ranking function associates with each program state a value from a well-founded domain. Intuitively, a state gets a smaller rank if it is "closer" to fulfilling the eventuality and gets a minimal rank when the eventuality is fulfilled.

We suggest an automated heuristic for choosing ranking functions. Based on these functions we then mark an abstract transition with D (*decreasing*) if every concrete transition associated with it definitely leads to a smaller rank. We mark it with I (*increasing*) if some concrete transition associated with it may lead to a larger rank.

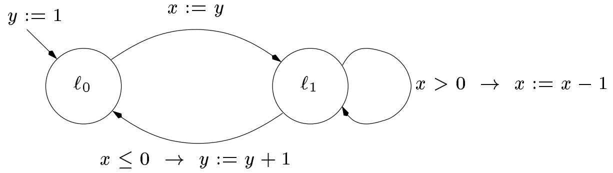
We can now exclude from the abstract model all paths that visit infinitely often D transitions but do not visit infinitely often I transitions. This is done by adding Streett acceptance conditions to the abstract model and apply a model checking algorithm that considers only the *fair* paths ([4, 12]), namely, those that satisfy the Streett conditions.

The excluded paths clearly do not correspond to any concrete path and therefore should not be considered for verification. The resulting abstract model is still an over-approximation of the concrete one, but is more precise. Since additional cycles were eliminated, the new model will satisfy more liveness properties. Thus, we will be able to verify more liveness properties of the concrete model.

The rest of the paper is organized as follows. Section 2 presents the abstract interpretation framework and the predicate abstraction approach using a simple example. The example also demonstrates the difficulty that arises when using abstraction for verifying liveness properties. Section 3 presents our heuristic for generating ranking functions and applies it to the example. Finally, Section 4 identifies directions for future research.

2 Abstraction

In this section we recall the main principles of abstraction of state-transition diagrams. The following small program will serve as a running example. The variables x and y are of type integer.



Suppose that we are interested in formally verifying that the program satisfies the temporal logic ([16, 11]) property $\mathsf{GFat\_}\ell_0$, which says that infinitely often, control must be at location ℓ_0. Obviously, we cannot establish this by finite-state model checking, as the part of the state space that is relevant to this property consists of infinitely many states. Namely, the state-transition graph induced by this program is $(\ell_0, \_, 1) \rightarrow (\ell_1, 1, 1) \rightarrow (\ell_1, 0, 1) \rightarrow (\ell_0, 0, 2) \rightarrow (\ell_1, 2, 2) \rightarrow (\ell_1, 1, 2) \rightarrow (\ell_1, 0, 2) \rightarrow \cdots$, consisting of states (ℓ, x, y), each giving the program location and the values of the two variables, and that are connected by transitions as generated by executing the program's transformations (statements) occurring along its edges. Note that in this case the state-transition graph is a linear structure; in general it may display branching behaviour and cycles.

One way to deal with this is to *abstract* the infinite state-transition graph that captures the program's behaviour into a finite, "abstract", structure which can then be model checked. This approach must obviously satisfy two criteria. First, it should come with a notion of *preservation*, that tells us which correctness properties (temporal logic formulae), if they can be established to hold for the abstract structure, hold for the original structure as well. Second, there must be a way to *construct* the abstract structure that circumvents explicitly building the infinite concrete structure as an intermediate step.

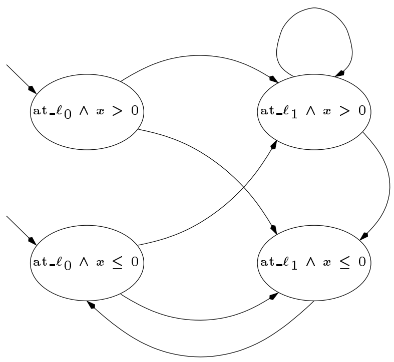
The notion of preservation we use here is based on a well-known result from automata theory, namely that the existence of a *simulation relation* between the states of two state-transition graphs guarantees that the traces (behaviours, executions) of one are included in those of the other. We use an instantiation of this where the simulation relation is indeed a function α from concrete to abstract states. Such an *abstraction function* induces a partitioning of the concrete state space, where an abstract state a is said to *describe* all concrete states which α maps to a. By requiring α to be a simulation relation, there must be an (abstract) transition between two abstract states a and b if there is a transition from some concrete state mapped to a to some concrete state mapped to b. Furthermore, in order to be able to interpret temporal logic formulae over abstract structures, we need to specify how the truth of propositions is defined in abstract states. We require that if a *literal* q (a literal is either a proposition or its negation) is defined to be true in abstract state a, then it must be true in all concrete states described by a. In addition, we require formulae to be brought in negation normal form before being evaluated over an abstract structure. These requirements suffice for the preservation of properties expressed in *universal* temporal logics, which do not use existential quantification over behaviours, see e.g. [6, 10]. Observe that the *truth* of formulae is preserved from the abstract to the concrete structure. However, if a formula does not hold over an abstract structure, this may be because the abstraction is too coarse. In other words, we may have *false negatives.*

In order to construct the abstract structure we use *abstract interpretation*. Here, the idea is that given a representation of the abstract states, the abstract transition relation is computed by symbolically interpreting the program's state transformations over these states ([7]). This will be illustrated below.

A topic that is drawing considerable attention recently is the identification of heuristics for the automatic generation of abstractions, given a program and a correctness property to verify. The quality of such heuristics is determined by the extent to which they produce abstract structures that are small enough to be manageable by a model checker, yet contain enough detail to allow to establish the property.

A popular approach is *predicate abstraction*, introduced in [13] and inspired by work on the automatic generation of invariants ([17, 3]) and on partition refinement ([9, 10]). In this method, the abstract states are represented by predicates that are formulae in some decidable logic that is supported by automated decision procedures. If in addition the state transformations performed by the program are expressed in the same logic, then the decision procedures can be used to automatically compute the abstract transition relation, i.e., to perform the abstract interpretation of the program's instructions over the predicates representing the abstract states, thus constructing the reachable part of the abstract state space by "symbolic execution". Examples of tools that offer (some of) these capabilities are STeP [18], InVeSt [2], and Bandera [1].

The predicates for the initial abstraction are usually derived from syntactic clues in the program text and the correctness property. For example, the abstract states are taken to be conjunctions of predicates expressing that control is at a certain point with the set consisting of all conditions occurring in the program, together with all predicates occurring in the property. We illustrate this by (manually) applying it to our example. The "control predicates" are $\mathrm{at\_}\ell_0$ and $\mathrm{at\_}\ell_1$, the two guards are $x > 0$ and $x \leq 0$, and the predicate in the correctness property is $\mathrm{at\_}\ell_0$ again. Together, these give the abstract states $\mathrm{at\_}\ell_0 \wedge x > 0$, $\mathrm{at\_}\ell_0 \wedge x \leq 0$, $\mathrm{at\_}\ell_1 \wedge x > 0$, and $\mathrm{at\_}\ell_1 \wedge x \leq 0$. These states and the abstract transitions induced are depicted below. Consider for example the abstract transition from $\mathrm{at\_}\ell_0 \wedge x > 0$ to $\mathrm{at\_}\ell_1 \wedge x \leq 0$. The predicate $\mathrm{at\_}\ell_0 \wedge x > 0$ puts no constraints on the value of y, so a concrete state like $(\ell_0, 5, -2)$ is described by it. There is a concrete transition from this state to $(\ell_1, -2, -2)$ (by performing the transformation $x := y$), which in turn is described by $\mathrm{at\_}\ell_1 \wedge x \leq 0$. Note that even in cases where not everything can be expressed in the decidable logic, the decision procedures can be used to produce overapproximations of the abstract structure, which may still contain valuable information.



The formula $\mathsf{GF}\mathrm{at\_}\ell_0$ does not hold in this model, the counterexamples being those executions that end up cycling in state $\mathrm{at\_}\ell_1 \wedge x > 0$. For a human it is easy to see that this is a false counterexample: The guarded command that is responsible for the transition from this abstract state to itself, $x > 0 \rightarrow x := x - 1$, can clearly not be executed infinitely often without leaving the state.

In addition to heuristics for constructing an initial abstraction, the method of predicate abstraction is characterised by a recipe for refining an abstract structure in cases as this, where false negatives are present. However, in the case of this example, it is not possible to get rid of this spurious loop by refining the predicates that make up the abstract states and keep the abstraction finite at the same time. In order to do so, predicates $x = n$ would have to be introduced for every value n that x may take, i.e., for every positive n, thus leading to an infinite-state structure.

3 Generating Ranking Functions

Instead of making the *invariants*, i.e. the abstract states, more precise, we need to define fairness or acceptance conditions that will exclude abstract paths that do not correspond to any concrete path. In [14], so-called progress monitors are introduced to this purpose. Here, we will attach the progress information more directly to the abstract structure, namely by labelling transitions with I (for "possibly increasing") or D (for "definitely decreasing"). These notions are relative to a ranking function mapping concrete states into some well-founded domain. If an abstract transition is labelled D then this means that along *all* corresponding concrete transitions, the value of the ranking function decreases strictly (this explains the "definite"). An I label means that along *some* corresponding concrete transition, the value of the ranking function *may* increase (hence the "possibly"). So, it is indeed the absence of an I label on an abstract transition that conveys the definite information that none of the corresponding concrete transitions increases the ranking. Clearly, an infinite abstract path can only have a corresponding infinite concrete path if it satisfies the following linear-time temporal formula which means "infinitely often decreasing implies infinitely often increasing".

$$\mathsf{GF}D \ \Rightarrow \ \mathsf{GF}I \tag{1}$$

Converting the formula to a Streett acceptance condition and adding the condition to the abstract model will identify as non-fair all paths that do not satisfy the formula. The model checking algorithm will then ignore these paths. These paths indeed should be ignored since they do not represent any real concrete behaviour.

Observe that *any* ranking function, mapping into *any* well-founded domain, may give useful progress information, as long as the D/I markings are correct. Indeed, multiple ranking functions[1] rf_i may be introduced, leading to multiple D_i/I_i markings, each coming with a Streett acceptance condition of the form (1). This is comparable to the case of finding abstract states as e.g. given by the predicate abstraction method: *Any* set of predicates may be useful in proving invariance properties, as long as the abstract transitions are computed correctly. Of course, there is a trade-off: just as the introduction of too many predicates may lead to a state explosion, the introduction of too many ranking functions may complicate model checking.

The discussion above brings up the following two questions.

1. Given a program, how can we automatically identify useful ranking functions?
2. Given a ranking function, how to automatically compute the D/I effect of a concrete transformation?

We propose a method that is in the same spirit as the idea of predicate abstractions. It relies on a heuristic to generate ranking functions, based on syntactic clues in a program. Then, decision procedures are used to compute the D/I markings.

Just like in the case of the heuristic for choosing a set of predicates in the case of predicate abstraction, also the ranking functions are chosen based on the *conditions* that occur in the program text. The idea is to view a condition as a goal or eventuality, something that must

[1] From now on we will assume the definition of the well-founded domain to be part of the definition of the ranking function.

eventually be reached, and to introduce a measure, ranging over a well-founded domain, that decreases as the program gets closer to fulfillment of this goal, and that hits the "bottom" of the domain as soon as the goal is reached. Consider our example program again. The condition $x \leq 0$ suggests the following ranking function (over $\mathbb{N}$):

$$rf_{x\leq 0} : \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leq 0 \end{cases}$$

As the condition $x \leq 0$ guards the transition from ℓ_1 to ℓ_0 in the program, the intuition behind viewing it as a goal is that we are interested in "eventually getting out of ℓ_1" — which is precisely what is needed in order to break the selfloop on state $\ell_1 \wedge x > 0$ in the abstract structure. The associated ranking function $rf_{x\leq 0}$ is positive when $x > 0$, and decreases as the program "gets closer" to $x \leq 0$. As soon as $x \leq 0$ has been reached, it is 0. Computing the effect of the transformations occurring in the program, it turns out that, in the abstract state graph, the selfloop on state $\ell_1 \wedge x > 0$ and the abstract transition from $\ell_1 \wedge x > 0$ to $\ell_1 \wedge x \leq 0$ are labelled with $D_{x\leq 0}$, and all other transitions with $I_{x\leq 0}$. It is not hard to see that under the condition $\mathsf{GF}D_{x\leq 0} \Rightarrow \mathsf{GF}I_{x\leq 0}$, the formula $\mathsf{GFat\_}\ell_0$ now holds on the abstract structure.

More generally, we suggest the following rules to generate ranking functions for a number of simple (in)equality conditions over integers; x is a program variable and N a constant. The ranking functions have been "scaled" so that their domain is always $\mathbb{N}$.

condition c	ranking function rf_c
$x \leq N$	$\begin{cases} x - N & \text{if } x > N, \\ 0 & \text{if } x \leq N \end{cases}$
$x < N$	$\begin{cases} x + 1 - N & \text{if } x \geq N, \\ 0 & \text{if } x < N \end{cases}$
$x \geq N$	$\begin{cases} N - x & \text{if } x < N, \\ 0 & \text{if } x \geq N \end{cases}$
$x > N$	$\begin{cases} N + 1 - x & \text{if } x \leq N, \\ 0 & \text{if } x > N \end{cases}$
$x = N$	$\lvert x - N \rvert$

We could also consider the other condition in our example program, $x > 0$, to generate a ranking function. According to the table, this would then be $1 - x$. In that case, the selfloop on state $\ell_1 \wedge x > 0$ would be marked with $I_{x>0}$, and hence it would not help in establishing the correctness property $\mathsf{GFat\_}\ell_0$.

In order to compute the D/I labels for a given ranking function, decision procedures may be used, again much in the same way they are used in the automatic generation of the transition relation. Also here, approximation may be used to circumvent undecidable or too-costly cases. In that case, one starts with an abstract structure in which all transitions are labelled with I and none with D. Only if it can be decided that a transformation does definitely not increase a ranking function, the corresponding I label may be removed. If in addition it can be guaranteed that the transformation causes a strict decrease in the ranking, a D label is added.

4 Reflections

To evaluate the usefulness of the heuristic we have suggested, it needs to be implemented and experimented with. Existing tools that incorporate facilities for automatically generating invariants, like STeP ([18]) or InVeSt ([2]), are probably convenient platforms for a prototype implementation. The reason is that our heuristic is closely related, both in aim and in structure, to such methods.

In order for the heuristic to be practically applicable, it is imperative that it is generalised to other data types beside integers, like arrays, queues, stacks, (linked) lists, tables, sets, etc. Case studies that we are currently undertaking indicate that such generalisations exist and are in many cases in terms of simple rules, like the table on page 6, based on the form of conditions occurring in a program.

Another issue that deserves further investigation is how to "refine" ranking functions or D/I labels in case they do not provide enough information to establish the property of interest. One possibility would be to associate D/I labels with *sequences* of transformations. For example, consider the loop **do** $x := x + 1; x := x - 2$ **until** $x \leq 0$. The ranking function generated by the termination condition is x if $x > 0$ and 0 otherwise. Labelling the individual statements would lead to an I label associated with $x := x + 1$ and a D label with $x := x - 2$, the result being that termination of the loop might still not be provable, even under the constraint $\mathsf{GF}D \Rightarrow \mathsf{GF}I$. However, the effect of the sequence $x := x + 1; x := x - 2$ is D, which could suffice for proving termination.

On the theoretical side, an interesting topic is the integration of the idea of "abstracting into ω-automata" into the theory of Abstract Interpretation ([7, 8]).

References

[1] The Bandera project. See `http://www.cis.ksu.edu/santos/bandera/`.

[2] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt : A tool for the verification of invariants. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, number 1427 in LNCS, Berlin, 1998. Springer.

[3] Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, number 1102 in LNCS, pages 323–335, Berlin, 1996. Springer.

[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, January 1986.

[5] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *11th International Conference on Computer Aided Verification (CAV'00)*, LNCS, Chicago, USA, July 2000.

[6] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[7] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[8] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, number 631 in LNCS, pages 269–295, Berlin, 1992. Springer-Verlag.

[9] Dennis Dams, Rob Gerth, and Orna Grumberg. Generation of reduced models for checking fragments of CTL. In Costas Courcoubetis, editor, *Computer Aided Verification*, number 697 in LNCS, pages 479–490, Berlin, 1993. Springer-Verlag.

[10] Dennis René Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, July 1996.

[11] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Formal Models and Semantic*, volume B of *Handbook of Theoretical Computer Science*, chapter 16, pages 995–1072. Elsevier/The MIT Press, Amsterdam, 1990.

[12] E.A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. 18th Hawaii International Conference on System Sciences*, Hawaii, 1985.

[13] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification*, number 1254 in LNCS, pages 72–83, Berlin, 1997. Springer.

[14] Yonit Kesten and Amir Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, To appear in special issue.

[15] R. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[16] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.

[17] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In U. Montanari, editor, *1st Intl. Conf. on Principles and Practice of Constraint Programming*, 1995.

[18] The Stanford Temporal Prover. See `http://rodin.stanford.edu/`.