Bonsai: Cutting Models Down to Size

Stefan Vijzelaar, Kees Verstoep, Wan Fokkink, and Henri Bal

VU University Amsterdam, The Netherlands
{s.j.j.vijzelaar,c.verstoep,w.j.fokkink,h.e.bal}@vu.nl

Abstract In model checking, abstractions can cause spurious results, which need to be verified in the concrete system to gain conclusive results. Verification based on multi-valued model checking can distinguish conclusive and inconclusive results, while increasing precision over traditional two-valued over- and under-abstractions. This paper describes the theory and implementation of multi-valued model checking for Promela specifications. We believe our tool Bonsai is the first four-valued model checker capable of multi-valued verification of parallel models, i.e. consisting of multiple concurrent processes. A novel aspect is the ability to only partially abstract a model, keeping parts of it concrete.

1 Introduction

The ubiquitous problem of state space explosion, i.e. a combinatorial blow-up of behaviour, is a central theme in the verification of systems. While abstraction can reduce the impact of state space explosion, it can also introduce spurious results [5]. By combining over- and under-abstraction, it is possible to identify abstract behaviour which is guaranteed to match the concrete behaviour of the system. This can be implemented using three-valued semantics [2,15]: properties can be either *true*, *false*, or *unknown*; any result which would have been spurious in the over- or under-abstraction is represented by the *unknown* value.

An elegant way to model abstract transitions is to use a four-valued logic [1]. The truth values of the logic form a bilattice [8], the elements of which are in both a truth ordering and an orthogonal information ordering. Operations of the logic map to operations over the truth ordering, while abstractions of the system can be mapped to operations over the information ordering. An added benefit of this strong relation between the logic and truth ordering is a natural definition of existing temporal logics in terms of lattice operations [16,3]. These definitions can be reused for other multi-valued logics [11], conceivably resulting from new abstraction or modelling techniques.

Techniques based on a four-valued logic have been successfully used in symbolic trajectory evaluation for verification of logical circuits [17], and abstract model checking of software [13]. In this paper we are interested in applying the multi-valued approach to concurrent software systems, for which to our knowledge there are no tools available at this point. We generalise the abstraction technique used in [12] to use operations in the information ordering of the bilattice, and implement this technique in a tool for concurrent processes.

Since we prefer to extend on existing work, we focus on concurrent Promela models as used by the SPIN [14] model checker. For these models we implement four-valued abstraction, combining two-valued over- and three-valued under-abstraction, in a tool called Bonsai. Abstractions are constructed using predicate abstraction [9], which is a special case of abstract interpretation [6].

The implementation is written in Java, and based on the SpinJa model checker [7] and the SMTInterpol satisfiability solver [4]. The two-valued semantics of the SpinJa model checker can be reused by decomposing the four-valued model checking problem into two, classical, two-valued problems [11] using the satisfiability solver for abstraction. This method can be extended to model check other higher-valued logics.

The paper is structured as follows. Section 2 gives an introduction to multi-valued model checking; it shows the four-valued logic used in our abstraction, a method for constructing the multi-valued abstraction, and the decomposition applied by our tool. In section 3 we detail the implementation and show that the decomposed problems can share results: it is not required to calculate two completely separate abstractions to get a multi-valued result. Section 4 demonstrates the tools effectiveness at some typical examples for abstraction, while in section 5 we conclude and consider future applications.

The long term goal of this tool is to investigate four-valued and other higher-valued logics for concurrent processes. Specifically logics which separately model steerable and unsteerable non-determinism can prove to be interesting: results of multi-valued abstract verification could be combined with runtime steering to guarantee correct execution of software for which verification would otherwise have been intractable.

2 Multi-valued model checking

2.1 Preliminaries

A lattice is a partially ordered ($\sqsupseteq$) set, in which any two elements have a least upper bound (supremum or join), and a greatest lower bound (infimum or meet). By induction, a non-empty finite lattice has a join and meet for each subset of elements. Therefore, the set as a whole is bounded, and has a greatest element (top or $\top$), and least element (bottom or $\bot$).

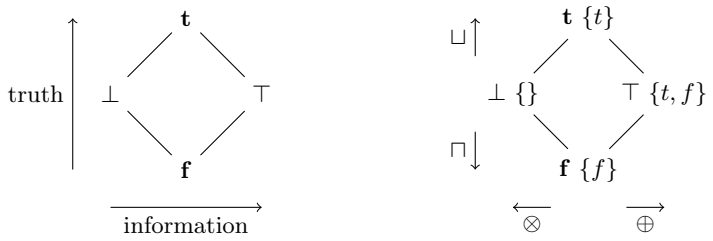
Lattices can be used to define quasi-boolean algebras which can be applied when verifying temporal properties. Model checking typically uses classical boolean logic: transitions between states either exist (are *true*) or do not exist (are *false*); and atomic propositions used by temporal properties either hold for a state (are *true*) or do not hold (are *false*). It is customary to only draw *true* transitions in a state space graph; missing transitions are assumed to be *false*.

The classical boolean logic used to verify properties can be described in the more general framework of lattice theory: a lattice consisting of two elements, with *true* being the supremum, and *false* being the infimum. The boolean conjunction and disjunction operations map respectively to the meet ($\sqcap$) and join ($\sqcup$) of lattice theory.

In multi-valued model checking, instead of classical boolean logic, more general quasi-boolean logics can be used. The truth values of a quasi-boolean logic are the elements of a finite distributive lattice. Conjunction and disjunction map to meet ($\sqcap$) and join ($\sqcup$) respectively, while negation ($\sim$) needs to adhere to De Morgan's laws and the law of double negation. Distributivity of the lattice ensures that the meet and join distribute over each other, similar to conjunction and disjunction in classical boolean logic.

The strong relation between boolean operations and lattice operations ensures that the verification of temporal properties remains the same for different quasi-boolean logics. Classical definitions of temporal properties can easily be translated to lattice operations, and are then applicable to the more general class of quasi-boolean logics instead of just classical boolean logic.

2.2 A lattice for under- and over-abstraction



(a) Orthogonal information ordering (b) Operations and interpretations

Figure 1: Multi-valued lattice for under- and over-abstraction

We can use a quasi-boolean logic, based on a lattice, to model both under- and over-abstraction at the same time. The interlaced bilattice [8] used for this purpose in [12] not only defines the required truth ordering of the logic, but also an orthogonal information ordering; see Fig. 1a. As a consequence of this additional ordering, the $\top$ and $\bot$ elements should not be interpreted as the top and bottom of the logic: they are used to model the top and bottom of the information ordering.

One way to characterise the additional two truth values is to interpret *bottom* ($\bot$) as neither true nor false, and *top* ($\top$) as both true and false. In other words, the elements of the information ordering can be seen as sets, which can contain an item for truth (t) and an item for falsity (f). Truth values no longer map to a single items of the set $\{t, f\}$, as is the case for classical logic, but to its subsets. This allows for values which contain none ($\bot$) or both ($\top$) of the elements in $\{t, f\}$, as can be seen in Fig. 1b.

We can apply this interpretation to the atomic propositions and transitions of a transition system, and by extension to temporal properties evaluated over

this system. Each of these concepts can be modelled using the same four-valued logic, but this does not mean that all truth values are meaningful for every context. Depending on the context, some values should not occur.

Atomic propositions can be *true* (**t**), *false* (**f**) or *unknown* ($\bot$). The *unknown* value for a proposition is represented as the absence of knowledge by using the bottom element of the information ordering ($\bot$). It can be used to express a loss of information due to abstraction: we conclude neither true nor false for an atomic proposition which has been assigned this value. Conversely, the value *top* ($\top$) will never be assigned to a proposition, since atomic propositions cannot be both true and false at the same time.

Transitions can be *may* transitions ($\bot$), *must* transitions ($\top$), both (**t**), or neither (**f**). This requires a more general definition of *may* and *must* transitions than is used by modal transition systems which only recognise *must* transitions as a subset of *may* transitions (e.g. [10]). We define *may* transitions as those transitions that are at least not false (i.e. of value $\bot$ or **t**), and *must* transitions as those transitions that are at least true (i.e. of value $\top$ or **t**). The use of the *top* ($\top$) value allows us to express that for a set of states, some states are reachable while others are not.

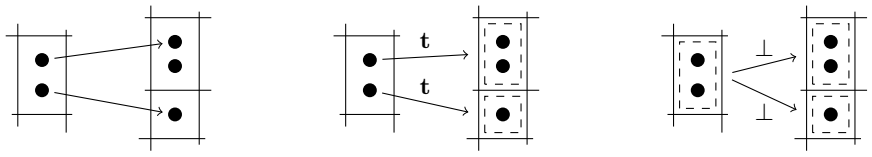
By extension of the atomic propositions and transitions of the system we can evaluate temporal properties over the system. Temporal properties use the same values as atomic propositions: *true* (**t**), *false* (**f**) or *unknown* ($\bot$). Similar to atomic propositions they express a property of a state: the reachability of behaviour from said state. Even though transitions can take on the value *top* ($\top$), this value should never result in a temporal property of the same value: a temporal property cannot be both true and false at the same time.

Note that while *bottom* and *top* behave similarly in the logic, i.e. when only using operations on the truth ordering, they are not interchangeable when taking into account operations on the information ordering, which we will be using when constructing abstractions. It is however possible to obtain a similar construction if also the operations used for the abstraction method are interchanged.

2.3 Multi-valued abstraction

Using the notion of an information ordering, it is no longer necessary to separately reason about under- and over-abstraction. It is possible to use a single generic multi-valued method of abstraction which captures both types of abstraction using operations on the information ordering. For this purpose we will be using the meet ($\otimes$) and join ($\oplus$) operations on the information ordering. See Fig. 1b for an overview of the operations on the truth and information ordering.

Assume we have defined equivalence classes over a set of concrete states of a Kripke structure, e.g. using predicates. Fig. 2a shows a concrete example system: source states are on the left and destination states on the right. The lines are used to indicate how predicates divide the concrete states into equivalence classes. An abstract state is formed by those concrete states which have the same evaluation for all predicates. To complete the abstraction we want to lift the transitions between concrete states to transitions between abstract states.

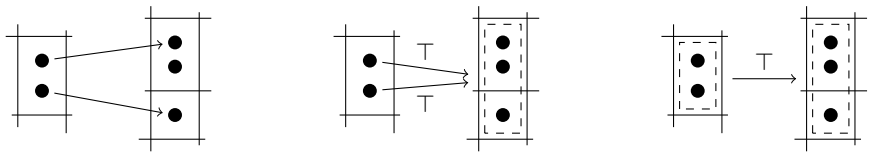


(a) Concrete system (b) Destination states (c) Source states

Figure 2: Abstraction with individual abstract states

Our abstraction method to lift transitions starts by applying an induction hypothesis. It is assumed that we are able to correctly express the behaviour of the abstract destination states; therefore, we can replace any transition to a concrete state by a transition to the abstract state it belongs to. The results are shown in Fig. 2b. The induction hypothesis ensures that we do not have to differentiate between constituent states of the abstract destination state, and any possible loss of information is caused solely by the abstraction of behaviour for the abstract destination state.

To complete the induction hypothesis, we need to determine the behaviour of the abstract source state. For this purpose we calculate the consensus of the concrete source states on the reachability of specific abstract destinations; we do this by using the meet of the information ordering ($\otimes$). Stated differently, since we will lose the ability to differentiate between concrete source states for a given abstract source state, the best we can do is to describe the behaviour they agree on. In the example of Fig. 2b each abstract destination state is reachable ($\mathbf{t}$) by one concrete source state, and unreachable ($\mathbf{f}$, not drawn) by the other. This gives us the value *bottom* ($\mathbf{t} \otimes \mathbf{f} = \bot$) for each abstract destination state, as can be seen in Fig. 2c above.



(a) Concrete system (b) Destination states (c) Source states

Figure 3: Abstraction with a set of abstract states

This method in itself would be sufficient to create a multi-valued abstract model, a model containing aspects of both under- and over-abstraction, but we can do better. To increase precision we can merge abstract states by using the meet operator of the information ordering ($\otimes$). Take two or more abstract destination states that have different valuations for one or more properties and

combine their valuations using the meet operator. The result is a merged abstract state with the value *bottom* ($\bot$) for any predicate the original abstract states did not agree on. It models the possibility of these predicates being either true or false: their actual value is unknown.

Using these merged abstract states, we can model the reachability of sets of abstract states. While concrete source states might not agree on the reachability of *individual* abstract destination states, they could still agree on the reachability of a *set* of abstract destination states. We create sets by merging abstract states, with the provision that the resulting sets are limited to a cartesian product of predicate values by construction. Transitions to merged abstract states model whether any individual abstract state in the set is reachable, but without specifying *which* particular states are reachable.

To calculate the correct value of a transition from a concrete state to a set of abstract states, we combine the information of the states in the set using the join operator of the information ordering ($\oplus$). Since we are interested in the reachability of the set of abstract states as a whole, we want to aggregate the knowledge we have for reachability of the individual abstract states. This can be contrasted to the meet operator of the information ordering ($\otimes$) which calculates the consensus. In Fig. 3b it can be seen how this gives us the value *top* ($\mathbf{t} \oplus \mathbf{f} = \top$), since the merged abstract state contains both reachable and unreachable abstract states.

The behaviour of abstract source states is calculated in the usual manner, using the meet operator ($\otimes$). This also applies to sets of abstract source states, since we cannot distinguish between abstract source states in a set any more than we can distinguish between concrete states in an abstract state. The best we can do is to describe the behaviour the abstract source states in the set agree on. Fig. 3c shows how this results in *top* ($\top \oplus \top = \top$) for the case of a single abstract source state as used in the example.

We can generalise the abstraction method by considering individual abstract states as sets containing just one abstract state. The resulting generic method constructs a multi-valued abstract system with transitions between sets of abstract states.

The generic method can be summarised as follows. Start by calculating the reachability of abstract destination states from concrete source states. Subsequently use the join operator ($\oplus$) to aggregate all transitions to members of the abstract destination set. Finally use the meet operator ($\otimes$) to reach consensus for all concrete states contained by members of the abstract source set. Repeat the last two steps for other abstract source and destination sets.

2.4 Multi-valued through classical model checking

A multi-valued model checking problem can be reduced to multiple classical model checking problems [11]. This is done by identifying the join-irreducible elements J in the lattice of truth values. (Join-irreducible elements are those elements, except for the bottom element, which cannot be expressed as the join of two other elements.) The multi-valued model checking problem for a temporal

property can then be split into $|J|$ classical model checking problems. For an LTL property φ, a trace π, and the partial lattice ordering $\sqsupseteq$, this gives the following identity:

$$[\varphi]_\pi = \bigsqcup_{j \in J} (j \sqcap ([\varphi]_\pi \sqsupseteq j))$$

All truth values can be expressed as a combination of join operations on join-irreducible elements, or more precisely as a join of those join-irreducible elements which are smaller than or equal to that specific truth value. The expression $[\varphi]_\pi \sqsupseteq j$ is either *true* or *false*, ensuring that $j \sqcap ([\varphi]_\pi \sqsupseteq j)$ is either j or *false*. The end result is a join over a join-irreducible value smaller than or equal to $[\varphi]_\pi$ and *false* otherwise, and since *false* has no influence on the join operation, we get $[\varphi]_\pi$.

The identity above allows us to evaluate inequalities over $[\varphi]_\pi$ and combine the results, instead of determining $[\varphi]_\pi$ directly. To calculate $[\varphi]_\pi \sqsupseteq j$, the cut operator $\Uparrow$ is introduced [11] to syntactically distribute the inequality over the temporal property (after which it can be evaluated using classical model checking over a modified model):

$$[\varphi \Uparrow j]_\pi = [\varphi]_\pi \sqsupseteq j$$

We assume LTL formulas to be in release positive normal form (PNF):

$$\varphi_n = \mathit{true} \mid \mathit{false} \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \mid \varphi_1 \,\mathsf{R}\, \varphi_2$$

This ensures only atomic propositions can be negated, and allows for a simple reduction. (The implicit universal quantification of an LTL formula could complicate this reduction [11], was it not for the fact that we check for language emptiness using the negated LTL formula. That is, we apply the reduction to the negated LTL formula in PNF.)

$$\begin{aligned}
\mathit{true} \Uparrow j &= \mathbf{t} \sqsupseteq j & \mathit{false} \Uparrow j &= \mathbf{f} \sqsupseteq j \\
p \Uparrow j &= p \sqsupseteq j & \neg p \Uparrow j &= \neg p \sqsupseteq j \\
(\varphi \wedge \psi) \Uparrow j &= (\varphi \Uparrow j) \wedge (\psi \Uparrow j) & (\varphi \vee \psi) \Uparrow j &= (\varphi \Uparrow j) \vee (\psi \Uparrow j) \\
(\varphi \,\mathsf{U}\, \psi) \Uparrow j &= (\varphi \Uparrow j) \,\mathsf{U}_{\sqsupseteq j}\, (\psi \Uparrow j) & (\varphi \,\mathsf{R}\, \psi) \Uparrow j &= (\varphi \Uparrow j) \,\mathsf{R}_{\sqsupseteq j}\, (\psi \Uparrow j)
\end{aligned}$$

This reduction leaves us with some inequalities over j, and the operators $\mathsf{X}_{\sqsupseteq j}$, $\mathsf{U}_{\sqsupseteq j}$ and $\mathsf{R}_{\sqsupseteq j}$. The values of $\mathbf{t} \sqsupseteq j$ and $\mathbf{f} \sqsupseteq j$ can be put directly into the property, while the inequalities $p \sqsupseteq j$ and $\neg p \sqsupseteq j$ will need to be encoded into the model. The semantics of the $\mathsf{X}_{\sqsupseteq j}$, $\mathsf{U}_{\sqsupseteq j}$ and $\mathsf{R}_{\sqsupseteq j}$ operators are identical to their LTL counterparts for a classical boolean logic by only considering transitions with a value $v \sqsupseteq j$ to be true. Keeping only those transitions in the model allows us to use the classical X, U and R operators.

In general, we can evaluate the reduced property $\varphi \Uparrow j$ for each $j \in J$ separately, by creating an appropriate transition system for each $j \in J$ respectively.

Instead of evaluating φ over multi-valued paths, we evaluate $\varphi \Uparrow j$ over classical paths, by only modelling whether literals and transitions are $\sqsupseteq j$. This is sufficient to evaluate the inequalities and $\mathsf{X}_{\sqsupseteq j}$, $\mathsf{U}_{\sqsupseteq j}$ and $\mathsf{R}_{\sqsupseteq j}$ operators introduced by the reduction.

3 Implementation

The theory presented above can be used to implement a multi-valued model checker by decomposing problems into multiple classical model checking problems. We implement our multi-valued model checker Bonsai on top of the SpinJa model checker [7], and use the SMTInterpol satisfiability solver [4] to construct the decomposition.

3.1 Modifications to SpinJa

To abstract a Promela model with Bonsai, predicates can be added directly to the Promela specification by using the special type `pred`. These predicates are then automatically used during the subsequent abstraction process. For example, to add the predicate $x < 4$ to the specification, we write `pred x < 4` in the declaration list of either the specification itself, or one of its processes. By allowing declarations in both the specification and its processes, predicates can be made either global for the whole specification or local to a specific process type; this allows predicates to reference both global and local variables without breaking scoping rules. Note that a local predicate can reference global variables.

The version of the SpinJa model checker we use, has been slightly modified to parse predicates in a similar way to standard variables. In the original implementation, variables are stored by the SpinJa parser in a VariableStore; this design is copied to store predicates in a PredicateStore. Predicates in the PredicateStore can reference variables in the VariableStore, but do not yet have a variable associated with them for storing the actual value of the predicate. The PredicateStore simply acts as a bookkeeping device for keeping track of which predicates have been added to the specification. Together with the introduction of the `pred` type, this is the only required modification to the SpinJa model checker.

3.2 Overview of the abstraction

Parsing a specification with the modified SpinJa model checker, creates a *promela model* containing multiple automata. These automata are object-based representations of the processes as defined in the Promela specification. Since automata are a type of program graph, we can use them to create abstract program graphs: one abstract automata for each concrete automata of the specification. Together they form an *abstract model* of the specification with respect to the given predicates.

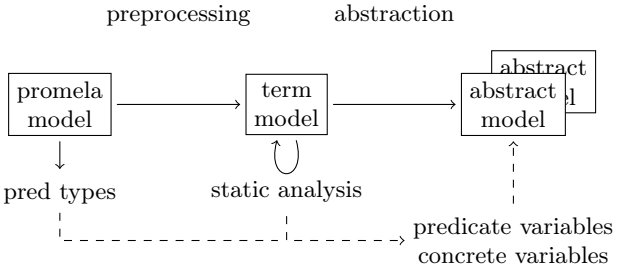


Figure 4: An overview of the implementation

This abstraction is done in two passes. Before we create the completed *abstract model*, we traverse the automata created by SpinJa one transition at a time. Transitions not influenced by abstraction (e.g. goto's) are copied directly into the *abstract model*, but other transitions (e.g. assignments) require preprocessing to create a *term transition* containing the metadata required for static analysis and abstraction. These *term transitions* are stored in a separate *term model*, which is a kind of scaffolding over the still incomplete *abstract model*.

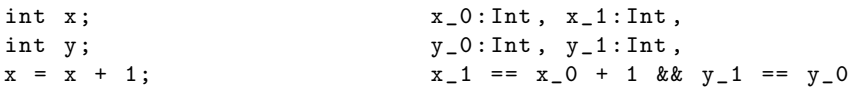
The *term model* has additional facilities for static analysis. In some cases it is useful to not abstract away from all concrete actions and variables of the original specification; for example, we might want to keep variables when they are part of the control flow. Static analysis can be used to determine which concrete variables, used by *term transitions*, need to be kept in the abstract automaton. Only after static analysis of the *term transition*s, do we add the actual variables to the *abstract model* for tracking concrete variables and predicates. To differentiate between them, all variable names in the *abstract model* have a prefix with their type. Predicates additionally have a unique number and a short descriptive string as a part of their name.

Using the *term model* we can generate the final over- and under-abstractions of the specification, which correspond to the two join-irreducible elements of the four-valued logic. For every *term transition* in the automaton, we generate two decision diagrams: one diagram for each abstraction. The diagrams are then encoded as transitions in an *abstract model* using only standard Promela if-statements and assignments. Together with the transitions already present, this completes the *abstract model*. For an overview see Fig. 4.

The resulting two *abstract models*, one for each join-irreducible element, can be compiled and verified using the default SpinJa tool stack. Each partial result indicates whether the multi-valued result is larger or equal to one of the join-irreducible elements. By combining them, we get a multi-valued result of the complete model checking problem.

3.3 Constructing SMT terms for transitions

Promela transitions consist of one or more actions, and each action can be modelled using two terms: a guard term, and an effect term. See Fig. 5 for an ex-



(a) Promela code (b) SMT term

Figure 5: Constructing an effect term

ample of an effect term being constructed from code: type definitions are shown for completeness. The guard term indicates whether the action is enabled, while the effect term models the relation between source and destination states. Even though only the first action in a transition is allowed to block, we cannot ignore the guard terms of the other actions. It depends on whether we are going to model the actions of a transition separately, or as one big abstract action; we might need the guard terms to detect blocking of subsequent actions in the transition, and report an error. The end result for a transition is a list of *term actions*, each containing a guard and effect term for the corresponding action.

To create terms for the individual actions we make use of the theories supported by the SMT solver, in this case the theory of integers. There is, however, no support for arrays in the solver we use, therefore any array encountered in the specification needs to be backed by separate term variables for each index; these are combined using an if-then-else construction based on the index term to model an array. At the same time we assume abstractions over indices may be unwanted, and keep track of any terms used as an index. One can argue that indices are at times part of the control flow of the program, i.e. a defining part of the program graph, and do not need be abstracted. Static analysis can then be applied to do a form of taint analysis: any variable used in an assignment to a concrete variable, also needs to be a concrete variable.

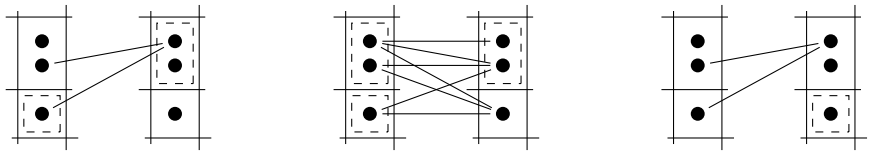
Using concrete values in a term can cause difficulties when abstracting transitions using the SMT solver. Concrete variables can store a large range of values; enumerating all of them in the SMT solver can make the abstraction intractable. As a first step, we ensure that assignments to concrete variables are never abstracted; such assignments are handled by concrete transitions, since we can assume that all referenced variables are also concrete. This assumption is guaranteed by the taint analysis, since concrete information cannot be created from abstract information.

Mixing concrete and abstract information is only allowed in specific cases. Either all variables used in a predicate are concrete, making the solution trivial since we can simply evaluate the predicate at runtime; or we require the concrete parts of the predicate to be somehow in a bounded domain, making the solution tractable. This is specifically the case when using a concrete value as the index of an array.

When constructing *term transitions* with concrete indices, we keep track of the possible range of these indices. This ensures that the SMT solver is bounded when enumerating all possible values of the index. This bound is over the complete expression which is used as an index, and can also be used to detect out-of-

bound conditions. Note that the use of concrete indices is useful when predicates reference specific indices in an array, but should only be used for small arrays, lest the abstraction would become intractable.

3.4 Abstracting SMT terms using predicates



(a) Under-abstraction (b) Negated effect term (c) Negated post-state

Figure 6: Under-abstraction by over-approximation

Given a list of *term actions* for a concrete transition, we want to generate an over- and under-abstraction of this transition. Each *term action* contains a guard and effect term. Since a guard is a simple boolean expression, we can easily over- and under-approximate this guard using the SMT solver. This leaves the effect term, for which we effectively want the following results: for the over-abstraction, we want to over-approximate the post-image of each abstract pre-state; and for the under-abstraction, we want to under-approximate the pre-image of each abstract post-state. While the over-approximation poses no problems, since over-approximation is a natural operation for an SMT solver, the under-approximation is not that straight forward.

We can calculate an under-approximation of a guard term, by negating the term, over-approximating it, and negating the result. For the effect term we want to under-approximate the pre-image of a specific abstract post-state; however, the pre-image is defined only implicitly by the combination of the effect term and the abstract post-state. As can be seen in Fig. 6, negating the effect term does not give the desired result, and negating the post-state only works when the effect relation is total.

One solution is to ensure that the effect relation is total by extending it, and to rely on an under-approximation of the guard term to filter out unwanted transitions. Then we could safely negate the post-state to calculate the required under-approximation of pre-states; however, this use of a guard in combination with a total function allows for a better solution. When trying to under-abstract the effect relation, we have the guarantee that all enabled concrete states have outgoing transitions: the under-approximated guard term reduces the domain of the effect relation to only enabled states and ensures it is total over this domain. In addition we only abstract individual deterministic transitions. We can use these facts to our advantage.

As a consequence of the above, the over- and under-abstraction can share results of the SMT solver. We start by over-approximating the effect function, which can be done using a single allSAT call to the SMT solver. This result can be shared between both abstractions. Next we respectively over- and under-approximate the guard and subsequently remove these pre-states from the over-approximation of the effect function. This gives us the two-valued over- and under-abstraction of the transition.

For an over-abstraction, the result can be used as-is to generate *abstract transitions*. A transition is created for each possible post-state. Pre-states related to this post-state can be identified using boolean conjunctions; their disjunction is the first action of the transition, and will act as a guard. The post-state can be constructed using a series of assignments, which will change the pre-state into the requested post-state; together with the guard action, these assign actions complete the transition. The construction of multiple transitions, allows the model checker to non-deterministically explore all possible post-states during verification.

For an under-abstraction, the result needs some additional processing. All post-states reachable from a given pre-state are flattened into a single new post-state. This is done by combining the predicate values of these post-states: if the post-states agree on the value of a predicate, that value is used; but if they disagree, the *unknown* value is used. This creates the most precise must transition for a fully specified pre-state, since we only abstract individual deterministic transitions. To include under-specified pre-states, we flatten sets of existing pre-states to form new pre-states, while we flatten their respective post-states to form a new post-state. By relating these new states, we can handle any possible pre-state in the under-abstraction. This results in a deterministic transition for each pre-state with outgoing transitions.

Since the method above already supports partial relations, we can further increase precision by also taking into account the guard term when abstracting the effect relation. We can actually ignore any concrete pre-state, which is not part of the guard. For an over-abstraction, the guard term can reduce the number of post-states reachable from a pre-state: removing impossible concrete transitions can reduce the number of may-transitions. For an under-abstraction, the guard term can reduce the number of post-states used by the flattening operations: preventing disagreement on predicate values can increase the amount of information in states after must-transitions.

3.5 Storing SMT results in decision diagrams

After abstracting the SMT terms, we need to store the results in some way. We also require support for different kinds of operations on these results, like the flattening operation described above. For this purpose we use a multi-valued decision diagram, allowing storage of predicate values and bounded concrete values. Predicate values are multi-valued: they can be *true*, *false* or *unknown*. Concrete values need to be part of some, preferably small, domain; this is not only to prevent large enumerations by the SMT solver, but also to allow tractable

negation of a result, e.g. for under-approximation. Finally we allow for the value *skip*, which is used to optimise assignments when predicate values are the same for both the pre- and post-state.

The diagrams we use are refined in multiple steps: we start by creating a generic decision diagram, which works for any type of value. It supports simple operations like union and intersection of diagrams. This is subsequently extended to a bounded term diagram by storing terms and their bounds at each node. Operations requiring multi-valued term information, like the flatten operation, are implemented at this level. Relations between sets of states are stored by creating nested term diagrams, which split the diagram into one outer and multiple inner diagrams: for each state in the outer diagram it contains an inner diagram containing related states. Finally we use assign diagrams to optimise the encoding of the diagram into actual transitions of the model; for example, values which do not change can be skipped when implementing a transition.

4 Experimental results

We use two mutual exclusion algorithms to demonstrate our implementation: Lamport's bakery algorithm, and Fischer's algorithm. Both algorithms use shared memory, and have potentially very large state spaces; they respectively model ticket numbers and discrete time, which can have domains of arbitrary size. These algorithms make typical examples for demonstrating the strengths of abstraction.

In Lamport's bakery algorithm, a process intending to enter its critical section picks a ticket number higher than any of the numbers used by other processes. It then waits until its number is the smallest of the waiting processes before starting its critical section. Due to concurrency, multiple processes can pick the same number, in which case the process id's are used as a tie-breaker.

For Fischer's algorithm, a single shared variable is used to keep track of reservations. A process reads the variable, and if it is zero, overwrites it with its own id. It then reads the variable for a second time, and can enter its critical section if its identity is still contained by the variable. Since concurrent processes can overwrite each others values, there is a timing constraint. It is required that after writing, a process waits a specified time before reading. This wait period needs to be longer than the time between reading zero and writing an id.

The abstractions used, map ticket numbers and discrete time values to smaller bounded domains. Enough information is retained to model check the algorithm. For Lamport's bakery algorithm, it is sufficient to map the relative ordering of ticket numbers, instead of their absolute values. Similarly, for Fischer's algorithm it is sufficient to keep track of remaining wait time, instead of absolute values of the clock and timers.

The concrete state spaces can be made arbitrarily large by increasing the maximum value for tickets or time after which the algorithm halts. In contrast, the abstract state space has a fixed size, irrespective of these values. In our tests

we demonstrate this effect by varying the maximum value, and showing its effect on the running time.

Tests are performed on a 2.66 GHz Intel Core 2 Duo, with 4GB of RAM. The Java virtual machine is given 2GB of heap space. Parameters for the SpinJa model checker are -m1000000 for the search depth and -DNOREDUCE to prevent partial order reduction. For the cases of more than 2^{14} ticket numbers or clock ticks, we use -m10000000 to increase the depth by a factor of 10 for the concrete model. During compilation we use -o3 to disable statement merging. Source code and model are available at `http://www.cs.vu.nl/~sjj.vijzelaar/spinja/`.

Abstracting the algorithms give typical examples of collapsing a large state space, an effect which shows clearly in the results (Tab. 1 and Tab. 2). Parsing and compiling of the abstract model takes significantly longer than for the concrete model, since parsing for the abstract model includes the construction of two abstract models. Any performance lost during construction, however, is easily gained during model checking. The state spaces of the concrete models grow with each increase of the model bounds. In the case of Fischer's algorithm, this even causes the model checker to run out of memory for more than 2^{15} clock ticks; the abstract model has no such problem.

	parse	compile	model checking (numbers)			
			2^{12}	2^{14}	2^{16}	2^{18}
abstract	12.35	5.81	0.33			
concrete	0.39	1.36	1.64	5.85	23.85	121.61

Table 1: Verifying Lamport's bakery algorithm (seconds)

	parse	compile	model checking (ticks)			
			2^{12}	2^{13}	2^{14}	2^{15}
abstract	30.41	8.20	1.47			
concrete	0.36	1.85	11.16	22.58	50.01	108.08

Table 2: Verifying Fischer's algorithm (seconds)

5 Conclusion

This paper gives an overview of the theory required to implement multi-valued verification on top of a classical two-valued model checker. We have used this

theory to implement four-valued abstract verification using the SpinJa model checker and SMTInterpol satisfiability solver. By doing so, we can now leverage the strength of the Promela language in modelling concurrent processes, and explore the benefits of multi-valued model checking in this context.

As far as we know, this tool is the first to implement multi-valued model checking of a quasi-boolean logic for concurrent processes. Additionally our model checker has the ability to apply abstraction to only parts of the concrete model. We want to apply these strengths to future research in the areas of runtime verification and execution steering.

References

1. N.D. Belnap. *Modern Uses of Multiple-Valued Logics*, pages 30–56. Reidel, 1977.
2. G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *CAV*, volume 1633 of *LNCS*, pages 274–287. Springer, 1999.
3. M. Chechik, B. Devereux, S.M. Easterbrook, and A. Gurfinkel. Multi-valued symbolic model-checking. *ACM TOSEM*, 12(4):371–408, 2003.
4. J. Christ, J. Hoenicke, and A. Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN*, volume 7385 of *LNCS*, pages 248–254. Springer, 2012.
5. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
7. M. de Jonge and T.C. Ruys. The SpinJa model checker. In *SPIN*, volume 6349 of *LNCS*, pages 124–128. Springer, 2010.
8. M. Fitting. Bilattices and the theory of truth. *Journal of Philosophical Logic*, 18:225–256, 1989.
9. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
10. O. Grumberg. 2-valued and 3-valued abstraction-refinement in model checking. In *Logics and Languages for Reliability and Security*, pages 105–128. IOS Press, 2010.
11. A. Gurfinkel and M. Chechik. Multi-valued model checking via classical model checking. In *CONCUR*, volume 2761 of *LNCS*, pages 263–277. Springer, 2003.
12. A. Gurfinkel and M. Chechik. Why waste a perfectly good abstraction. In *TACAS*, volume 3920 of *LNCS*, pages 212–226. Springer, 2006.
13. A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A software model-checker for verification and refutation. In *CAV*, volume 4144 of *LNCS*, pages 170–174. Springer, 2006.
14. G.J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.
15. M. Huth, R. Jagadeesan, and D.A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *ESOP*, volume 2028 of *LNCS*, pages 155–169. Springer, 2001.
16. B. Konikowska and W. Penczek. Reducing model checking from multi-valued CTL* to CTL*. In *CONCUR*, volume 2421 of *LNCS*, pages 226–239. Springer, 2002.
17. C.J.H. Seger and R.E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.