In AMAST'98, vol. 1548 of LNCS,
pp. 28-41, Springer-Verlag, Dec. 1998.

Visual Abstractions for Temporal Verification ⋆

Zohar Manna,
Anca Browne, Henny B. Sipma and Tomás E. Uribe

Computer Science Department
Stanford University
Stanford, CA. 94305-9045
manna@cs.stanford.edu

Abstract. *Generalized Verification Diagrams* combine deductive and algorithmic verification to establish general temporal properties of finite- and infinite-state reactive systems. The diagram serves as an abstraction of the system. This abstraction is deductively justified and algorithmically model checked. We present a new simple class of verification diagrams, using Müller acceptance conditions, and show how they can be used to verify general temporal properties of reactive systems.

1 Introduction

Reactive systems maintain an ongoing interaction with their environment, and include discrete, real-time and hybrid systems. *Deductive verification* is based on verification rules, which reduce the system validity of a temporal property to the general validity of first-order *verification conditions*. This methodology is complete relative to the underlying first-order reasoning, and allows the verification of a wide range of infinite-state systems. However, the proofs can be difficult to construct and understand, particularly as the complexity of the system increases.

Verification Diagrams provide a graphical representation of a deductive proof, summarizing the necessary verification conditions, and are therefore easier to construct and understand. *Generalized Verification Diagrams* extend them to be applicable to arbitrary temporal properties, replacing the well-formedness check on the diagram by a finite-state model checking step.

Diagrams can also be seen as an *abstraction* of the system, where properties of the diagram are guaranteed to hold for the system as well. The diagram represents the aspects of the system relevant to the property to be proved, and serves as an intermediary between the system and the property. To prove that a system $\mathcal{S}$ satisfies φ, we can find a diagram Ψ such that

$$\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi) \ ,$$

⋆ This research was supported in part by the National Science Foundation under grants CCR-95-27927 and CCR-9804100, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, ARO under grants DAAH04-95-1-0317, DAAH04-96-1-0122 and DAAG55-98-1-0471, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

where $\mathcal{L}(\mathcal{S})$ is the set of computations of $\mathcal{S}$, $\mathcal{L}(\Psi)$ is the set of computations of the diagram, and $\mathcal{L}(\varphi)$ is the set of models of φ.

The inclusion $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi)$ is proved deductively, by establishing verification conditions, and is equivalent to proving the correctness of an abstraction of $\mathcal{S}$. On the other hand, $\mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi)$ can be proved algorithmically, viewing the diagram as a finite ω-automaton. As ω-automata, diagrams can capture a class of properties that is strictly more general than those expressible in linear-time temporal logic.

Verification diagrams are thus a way of combining model checking and deductive verification. As abstractions, diagrams can be re-used in proofs of different properties, and provide visual documentation of the behaviors of the system.

In this paper we present a new, simple version of Generalized Verification Diagrams, based on Müller acceptance conditions. The Streett acceptance conditions of [BMS95] are equally expressive and more concise, but the new presentation allows a simpler, alternative definition. We show how these diagrams can be viewed as system abstractions, and used to verify general temporal properties of reactive systems.

Outline: Section 2 presents the basic background material. Section 3 presents the new class of Generalized Verification Diagrams. We discuss some practical issues in Section 4, and present related work in Section 5. In Section 6 we conclude by briefly describing our implementation of these methods in the Stanford Temporal Prover, STeP.

2 Background

2.1 Specifying Systems and Properties

We represent reactive systems as *fair transition systems* [MP95]. A fair transition system $\langle \Sigma, \Theta, \mathcal{T} \rangle$ is given by a set of states Σ, an *initial condition* Θ, which is a subset of Σ, and a set of *transitions* $\mathcal{T}$, each of which is a binary relation over Σ, describing how the system can move from one state to the next.

In our framework, we assume an *assertion language* based on first-order logic. The set Σ of possible system states is defined by a finite set of *system variables* $\mathcal{V}$; each transition τ is described by its *transition relation* $\rho_\tau(\mathcal{V}, \mathcal{V}')$, an assertion over the system variables $\mathcal{V}$ and a set of *primed variables* $\mathcal{V}'$ indicating their next-state values. Similarly, Θ can be expressed as an assertion over the system variables. We assume that $\mathcal{T}$ includes an *idling transition*, whose transition relation is $\mathcal{V} = \mathcal{V}'$. We use the standard triple notation for verification conditions,

$$\{\varphi\}\, \tau \, \{\psi\} \stackrel{\text{def}}{=} (\varphi(\mathcal{V}) \wedge \rho_\tau(\mathcal{V}, \mathcal{V}')) \rightarrow \psi(\mathcal{V}') \ \ .$$

A *run* of $\mathcal{S}$ is an infinite sequence of states $s_0, s_1, \ldots$, where s_0 satisfies Θ and for every s_i there is a transition $\tau \in \mathcal{T}$ such that (s_i, s_{i+1}) satisfy ρ_τ. In this case we say that τ is *taken* at s_i. The *enabled* predicate characterizes the set of states at which a transition τ can be taken:

$$\mathit{enabled}(\tau) \stackrel{\text{def}}{=} \exists \mathcal{V}'. \rho_\tau(\mathcal{V}, \mathcal{V}') \ \ .$$

$$\textbf{local}\ \ x, y : \textbf{integer where}\ x = 0\ \wedge\ y = 0$$

$$\begin{bmatrix} \textbf{loop forever do} \\ \quad \begin{bmatrix} \ell_0\!: \ \textbf{awaitc}\ x = 0 \\ \ell_1\!: \ y := y + 1 \end{bmatrix} \end{bmatrix} \quad \| \quad \begin{bmatrix} \textbf{loop forever do} \\ \quad \begin{bmatrix} m_0\!: \ \textbf{while}\ x > 0\ \textbf{do} \\ \quad \begin{bmatrix} m_1\!: \ x := x - 1 \end{bmatrix} \\ m_2\!: \ x := y \end{bmatrix} \end{bmatrix}$$

–P1– –P2–

Fig. 1. Program LOOPS

Transitions can be marked as *just* or *compassionate*. Just (or *weakly fair*) transitions cannot be continuously enabled without ever being taken. Compassionate (or *strongly fair*) transitions cannot be enabled infinitely often without being taken. Every compassionate transition is also just. A *computation* is a run that satisfies these fairness requirements.

Properties of systems are expressed as formulas in *linear-time temporal logic* (LTL). *Assertions*, or state-formulas, are first-order formulas with no temporal operators, and can include quantifiers. *Temporal formulas* are constructed from assertions, boolean connectives, and the usual *future* ($\Box, \Diamond, \bigcirc, \mathcal{U}, \mathcal{W}$) and *past* ($\boxminus, \diamondminus, \ominus, \mathcal{B}, \mathcal{S}$) temporal operators [MP95]. A *model* of a temporal property φ is an infinite sequence of states $s_1, s_2, \ldots$ that satisfies φ. For a system $\mathcal{S}$, we say that φ is *$\mathcal{S}$-valid* if all the computations of $\mathcal{S}$ are models of φ.

Example 1. Figure 1 shows program LOOPS, written in the Simple Programming Language (SPL) of [MP95]. SPL programs can be naturally translated into corresponding fair transition systems, following the semantics of each of the SPL constructs. To each process corresponds a *control variable*. For LOOPS, the control variables for processes P1 and P2 range over locations $\{\ell_0, \ell_1\}$ and $\{m_0, m_1, m_2\}$. The assertions m_i and ℓ_j are used to indicate the control location for each process, so the initial condition is:

$$\Theta : \ell_0 \wedge m_0 \wedge x = 0 \wedge y = 0\ \ .$$

The **awaitc** transition ℓ_0 is assumed to be a compassionate variant of the just **await** statement from [MP95]. That is, if control resides at ℓ_0 and $x = 0$ infinitely often, then the transition must eventually be taken. All other transitions are assumed to be just. Program LOOPS is an infinite-state system, since the variables x and y can grow beyond any bound. We will show that

$$\varphi : \Box(y > 0 \rightarrow \Diamond \Box(y \geq M))$$

is valid over this program, for any $M > 1$. That is, if $y > 0$ then eventually y will always be greater than the arbitrary constant M. □

We distinguish between *safety* and *progress* properties [MP95]. Intuitively, a safety property means that a particular class of "bad states" will never be reached. For example, *invariance formulas*, of the form $\square p$ for an assertion p, and *wait-for formulas*, of the form $\square(p \rightarrow q_1 \mathcal{W} \ldots \mathcal{W} q_n)$, express safety properties. Progress properties, on the other hand, state that certain states will eventually be reached. While safety properties do not depend on the fairness constraints of the system, progress properties do require fairness.

A binary relation $\succ$ is *well-founded* over a domain $\mathcal{D}$ if there are no infinite sequences of elements $e_1, e_2, \ldots$ in $\mathcal{D}$ such that $e_1 \succ e_2 \succ \ldots$ We write $x \succeq y$ iff $x \succ y$ or $x = y$. A *ranking function* δ is a mapping from system states into a *well-founded domain* $(\mathcal{D}, \succ)$.

2.2 Deductive Verification

Verification rules reduce the validity of a given temporal property over a given system $\mathcal{S}$ to the general validity of a set of first-order *verification conditions*. For example, the *general invariance rule*, which proves a property of the form $\square p$ for an assertion p, requires finding an assertion φ such that the following verification conditions are valid: (1) $\varphi \rightarrow p$ (that is, φ strengthens p), (2) $\Theta \rightarrow \varphi$ (that is, φ holds initially), and (3) $\{\varphi\}\, \tau\, \{\varphi\}$ for each transition $\tau \in \mathcal{T}$ (that is, φ is preserved by all transitions). Other verification rules are available for proving different classes of temporal properties [MP95].

3 Temporal Verification Diagrams

Verification diagrams were introduced by Manna and Pnueli [MP94] as a graphical representation of the verification conditions needed for a deductive proof. As with verification rules, different classes of diagrams are used to prove different classes of temporal properties.

Verification diagrams are generalized in [BMS95] to be applicable to arbitrary temporal properties, and shown to be a *complete* proof method for general (state-quantified) temporal formulas, relative to the reasoning required to establish verification conditions. The Generalized Verification Diagrams we describe below differ in presentation from those in [BMS95,BMS96], but the underlying notions remain the same.

A *Generalized Verification Diagram* (GVD) for a system $\mathcal{S}$ is a directed graph whose nodes are labeled by assertions, where a subset of the nodes is marked as *initial*. The assertion labeling a node n is indicated by $\mu(n)$. For a set of nodes $S : \{n_1, \ldots, n_k\}$, we define

$$\mu(S) \stackrel{\text{def}}{=} \mu(n_1) \vee \ldots \vee \mu(n_k) \ ,$$

where $\mu(\{\}) = \mathit{false}$. For a node n, the set of successor nodes of n is $\mathit{succ}(n)$.

A *run* of a diagram is a sequence $\sigma : s_0, s_1, \ldots$ of states of $\mathcal{S}$ such that there is an associated path $\pi : n_0, n_1, \ldots$ through the diagram, where n_0 is an initial node and for each $i \geq 0$, the state s_i satisfies $\mu(n_i)$.

We use an *acceptance condition* to define the limit behavior of the diagram. The theory of automata on infinite words (ω-automata) provides several types of acceptance conditions [Tho90]. For simplicity, here we choose *Müller* acceptance conditions. These are equally expressive as the more concise *Streett* acceptance conditions used in [BMS95], but allow a more intuitive presentation.

For an infinite path π through a GVD, let $inf(\pi)$ be the set of nodes that appear infinitely often in π. A *Müller acceptance condition* $\mathcal{F}$ is a set *of sets* of nodes. A path π is *accepting* if $inf(\pi) \in \mathcal{F}$. Note that an infinite path must eventually remain in a *strongly connected subgraph* (SCS), so an acceptance condition can always be expressed as a set of diagram SCS's. A *computation* of a diagram Ψ is a run of Ψ that has an associated accepting path. The set of all computations of Ψ is $\mathcal{L}(\Psi)$.

3.1 $(\mathcal{S}, \Psi)$ Verification Conditions

Associated with a GVD Ψ and a system $\mathcal{S}$ are verification conditions that, when valid, ensure that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi)$. In this case, we say that Ψ is $\mathcal{S}$-*valid*.

- **Initiation:** Every initial state of $\mathcal{S}$ should be covered by some initial diagram node, that is,
$$\Theta \rightarrow \mu(I) \ ,$$
where I is the set of initial diagram nodes. This implies that every run of $\mathcal{S}$ can start at some initial node of Ψ.
- **Consecution:** For every node n and every transition τ, there is a successor node that can be reached by taking τ (if τ can be taken at all), that is,
$$\mu(n) \wedge \rho_\tau \ \rightarrow \ \mu'(succ(n)) \ .$$
Here, $\mu'(succ(n))$ is the result of replacing each system variable x in $\mu(succ(n))$ by x'.

Together, these two conditions imply that every run of $\mathcal{S}$ can remain within Ψ:

Proposition 1. *If a diagram Ψ satisfies the initiation and consecution requirement for a system $\mathcal{S}$, then the runs of $\mathcal{S}$ are a subset of the runs of Ψ.*

Thus, once the above verification conditions are proved, we can conclude that any safety property of Ψ also holds for $\mathcal{S}$.

To preserve progress properties, a second set of verification conditions ensures that every computation of the system can follow an accepting path in the diagram, that is, can always eventually remain in an accepting SCS. Thus, if an SCS S is not accepting, we must show that computations can always leave S, or cannot stay in S forever.

For an SCS S, a *tail*(S)-*computation* is a system computation that has a corresponding path π in the diagram such that $inf(\pi) = S$. An SCS is called *transient* if every *tail*(S)-computation can leave S (so it is also a *tail*(S')-computation, for an SCS $S' \neq S$).

We want to show that every non-accepting SCS is transient. An SCS can be shown to be transient in one of the following three ways:

- **Just exit:** An SCS S has a *just exit*, if there is a just transition τ such that the following verification conditions are valid: for *every* node $m \in S$,

$$\mu(m) \;\rightarrow\; \mathit{enabled}(\tau) \qquad \text{and} \qquad \mu(m) \wedge \rho_\tau \;\rightarrow\; \mu'(\mathit{succ}(m) - S) \;\; .$$

  This means that τ is enabled and can leave the SCS at all nodes. We say that τ is the *just exit transition* for S.
- **Compassionate exit:** An SCS S has a *compassionate exit*, if there is a compassionate transition τ such that the following verification conditions are valid: for *every* node $m \in S$,

$$\mu(m) \;\rightarrow\; \neg\mathit{enabled}(\tau) \qquad \text{or} \qquad \mu(m) \wedge \rho_\tau \;\rightarrow\; \mu'(\mathit{succ}(m) - S) \;\; ,$$

  and for *some* node $n \in S$, τ is enabled at n:

$$\mu(n) \;\rightarrow\; \mathit{enabled}(\tau) \;\; .$$

  This means that for every node in S, either τ is disabled or τ can lead out of S, and there is at least one node n where τ can indeed leave S. We say that n is the *exit node* and τ is the *compassionate exit transition* for S.
- **Well-founded SCS:** An SCS $S : \{n_1, \ldots, n_k\}$ is *well-founded* if there exist ranking functions $\{\delta_1, \ldots, \delta_k\}$, where each δ_i maps the system states into elements of a well-founded domain $(\mathcal{D}, \succ)$, such that the following verification conditions are valid: there is a *cut-set*[1] E of edges in S such that for all edges $(n_1, n_2) \in E$ and every transition τ,

$$\mu(n_1) \wedge \rho_\tau \wedge \mu'(n_2) \;\rightarrow\; \delta_1(n_1) \succ \delta_2'(n_2) \;\; ,$$

  and for all other edges $(n_1, n_2) \notin E$ in S and for all transitions τ,

$$\mu(n_1) \wedge \rho_\tau \wedge \mu'(n_2) \;\rightarrow\; \delta_1(n_1) \succeq \delta_2'(n_2) \;\; .$$

  This means that there is no $\mathit{tail}(S)$-computation: it would have to traverse at least one of the edges in E infinitely often, which contradicts the well-foundedness of the ranking functions.

We say that S has a *fair exit* if it has a just or a compassionate exit. Combined with consecution, the fair exit verification conditions ensure that a $\mathit{tail}(S)$-computation can always follow a path that leaves S. Any run of the system that is *forced* to stay within an SCS with a fair exit must be unfair. If S is well-founded, there can be no $\mathit{tail}(S)$-computations. We can now claim:

Proposition 2. *If a GVD Ψ for a system $\mathcal{S}$ satisfies the initiation and consecution requirements, and all non-accepting SCS's have a fair exit or are well-founded, then $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi)$, that is, Ψ is $\mathcal{S}$-valid.*

[1] A cut-set of an SCS S is a set of edges E such that every loop in S contains some edge in E (that is, the removal of E disconnects S).

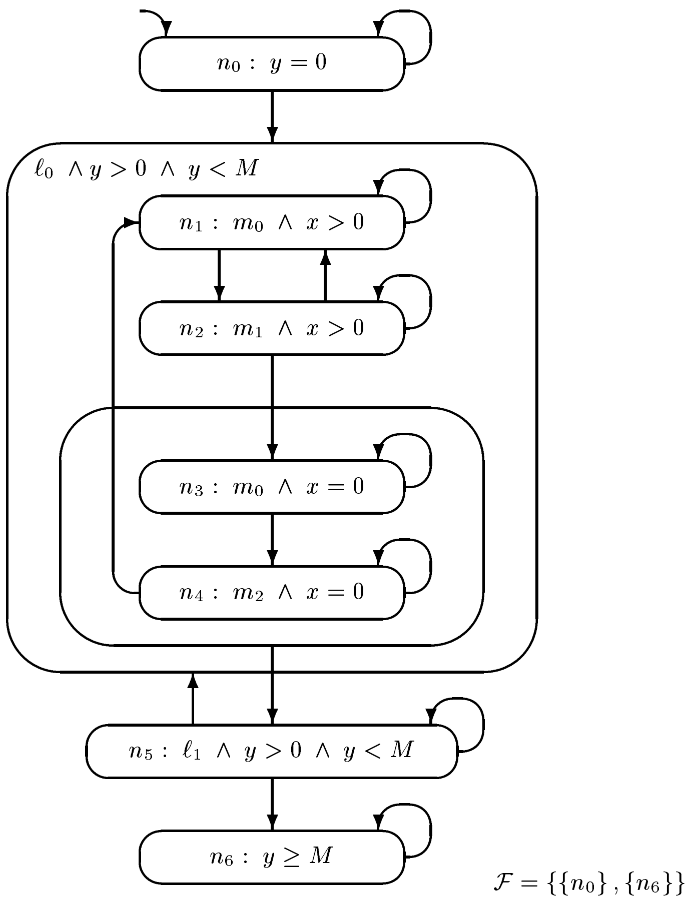

Fig. 2. A GVD Ψ for program LOOPS and property $\varphi : \square(y > 0 \rightarrow \diamond \square(y \geq M))$

Thus, to show that a given diagram Ψ is $\mathcal{S}$-valid, the user must prove initiation and consecution, and specify, for each non-accepting SCS, one of the following:

1. a just exit transition τ;
2. a compassionate exit transition τ and its exit node n; or
3. well-founded ranking functions δ_i and a cut-set E that prove that the SCS is well-founded.

Example 2. Figure 2 shows a GVD for the program LOOPS of Figure 1. The only initial node is n_0. M is a constant, where we assume $M > 1$. *Encapsulation conventions*, based on those of Statecharts [Har87], are used to make diagrams more succinct. Nodes n_3 and n_4 are part of a *compound node* which, together with n_1 and n_2, is part of a larger compound node. An assertion that labels a compound node is added, as a conjunct, to its subnodes. Edges leaving (entering) a compound node are interpreted as leaving (entering) all of its subnodes.

The runs of the diagram include all runs of program LOOPS. Initiation holds, since Θ implies $y = 0$. The consecution conditions can also easily be proved: for instance, the only transitions enabled at node n_1 are the idling transition and transition m_0, which when taken always leads to n_2. However, the diagram has runs that, for example, stay in node n_2 forever. These are not computations of the program, because they are not fair.

The diagram acceptance condition allows the transfer of progress properties from the diagram to the system, if the diagram is shown to be $\mathcal{S}$-valid. The weakest acceptance condition that lets us prove

$$\varphi : \Box(y > 0 \rightarrow \Diamond \Box(y \geq M))$$

is $\mathcal{F} : \{\{n_0\}, \{n_6\}\}$. The presence of $\{n_0\}$ means that computations of the diagram can stay at n_0 with $y = 0$ and never reach $y > 0$. The inclusion of $\{n_6\}$ means that once a diagram computation leaves the safe haven of n_0, it must be able to reach n_6 and stay there.

Thus, to prove that the diagram is $\mathcal{S}$-valid, we must show that all other SCS's are transient:

- It is straightforward to show that $\{n_1\}$, $\{n_2\}$, $\{n_3\}$, $\{n_4\}$ and $\{n_5\}$ have just exit transitions m_0, m_1, m_0, m_2 and ℓ_1, respectively.
- The SCS $\{n_1, n_2, n_3, n_4\}$ has compassionate exit transition ℓ_0, with exit node n_3 or n_4. Transition ℓ_0 is always enabled at n_3 and n_4, leading out of the SCS, and is and disabled at n_1 and n_2.
- The SCS $\{n_1, n_2\}$ can be shown to be well-founded with ranking function $\delta_i : x$ at both nodes. The value of x, always positive in this SCS, decreases along the edge $\langle n_2, n_1 \rangle$, which is a cut-set for the SCS, and does not change along the other edges.
- The remaining SCS's are of the form $\{\langle n_1, n_2, n_3, n_4 \rangle, n_5\}$, where $\langle n_1, \ldots, n_4 \rangle$ stands for any nonempty subset of $\{n_1, \ldots, n_4\}$. They can be shown to be well-founded using the ranking function $\delta_i : M - y$ at all nodes. This is well-founded, since $M > y$ within these SCS's. For each SCS, the set of edges that leave n_5 is the cut-set. These edges can only be traversed by transition ℓ_1, which increases y and thus decreases the well-founded order. The transitions on all other edges preserve the value of y. $\square$

3.2 (Ψ, φ) Property Satisfaction

Section 3.1 describes verification conditions that prove that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi)$, that is, that the diagram defines a correct abstraction of the system. To prove the $\mathcal{S}$-validity of a property φ, it remains to show that all the computations of the diagram are models of φ, that is,

$$\mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi) \ .$$

This check can be performed using standard ω-automata model checking, if we can relate the nodes of the diagram with the *atomic assertions* in φ (the ones with no subformulas other than themselves). As in [BMS95], we do this using a *propositional labeling*, where a diagram node n can be labeled with a boolean combination b of atomic assertions of φ, if

$$\mu(n) \rightarrow b$$

is valid. Given a propositional labeling, a GVD Ψ can be seen as a finite-state ω-automaton $\Psi^{\mathcal{A}}$, where each node is a state of the automaton, labeled with the given propositions. The property itself can be seen as an abstract property $\varphi^{\mathcal{A}}$ over its propositions. For instance, the atomic assertions in $\varphi : \Box(y > 0 \rightarrow \Diamond \Box(y \geq M))$ are $p : y > 0$ and $q : y \geq M$, which are now regarded as propositions. The abstract property is then $\varphi^{\mathcal{A}} : \Box(p \rightarrow \Diamond \Box q)$.

In most cases, the node label justification is trivial, since the diagram is drawn with φ in mind and the atomic assertions of φ are usually already present in the node assertions (see Section 4).

Example 3. Consider again the GVD for program LOOPS in Figure 2. Let $p : y > 0$ and $q : y \geq M$. Then we can label the nodes as follows:

$$\begin{array}{ll} n_0 : & \neg p \quad \text{(proving } y = 0 \rightarrow \neg(y > 0)) \\ n_1, n_2, n_3, n_4, n_5 : & \mathit{true} \text{ (no label is needed for these nodes)} \\ n_6 : & q \end{array}$$

The resulting ω-automaton $\Psi^{\mathcal{A}}$ is shown in Figure 3. The abstract property is $\varphi^{\mathcal{A}} : \Box(p \rightarrow \Diamond \Box q)$. We can now use finite-state model checking to automatically establish that $\Psi^{\mathcal{A}}$ satisfies $\varphi^{\mathcal{A}}$ [Kur94,BCM+92]. Thus, we have proved that $\varphi : \Box(y > 0 \rightarrow \Diamond \Box(y \geq M))$ is valid for program LOOPS. □

From an abstraction point of view, the propositional labeling ensures that the abstract properties model checked for $\Psi^{\mathcal{A}}$ imply the desired properties of the original system $\mathcal{S}$. The diagram can then be seen as a *weakly property-preserving abstraction* of the system $\mathcal{S}$ [Uri98].

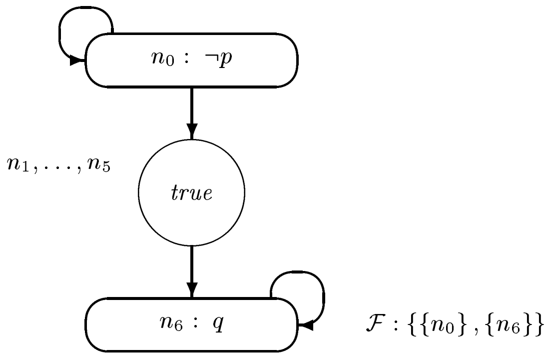

Fig. 3. Automata $\Psi^{\mathcal{A}}$, a propositional labeling of diagram Ψ of Figure 2

Once a diagram Ψ is proved to be a correct abstraction, i.e. $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Psi)$, it can be used to transfer any property that can be model checked for the diagram over to the system $\mathcal{S}$.

Example 4. The diagram from Figure 2 also lets us prove the $\mathcal{S}$-validity of

$$\Box((\ell_0 \ \wedge \ y > 0 \ \wedge \ y < M) \ \rightarrow \Diamond\, x = 0) \ .$$

In this case, we let $p : \ell_0$, $q : y > 0$, $r : y < M$ and $s : x = 0$, and label the nodes as follows:

$$\begin{array}{lll}
n_0 : & \neg q & (\text{proving } y = 0 \rightarrow \neg(y > 0)) \\
n_1, n_2 : & p \wedge q \wedge r & \\
n_3, n_4 : & s & \\
n_5 : & \neg p & (\text{proving } \ell_1 \rightarrow \neg \ell_0) \\
n_6 : & \neg r & (\text{proving } y \geq M \rightarrow \neg(y < M))
\end{array}$$

We can now, as before, model check $\varphi^{\mathcal{A}} : \Box((p \wedge q \wedge r) \ \rightarrow \ \Diamond\, s)$ over the resulting ω-automaton $\Psi^{\mathcal{A}}$. □

4 GVD Templates

As mentioned earlier, [MP94] provide different types of diagrams, depending on the type of temporal property being proved. In particular, INVARIANCE diagrams prove properties of the form $\Box\, p$, for an assertion p; WAIT-FOR diagrams prove properties of the form $\Box(p \rightarrow q_m \,\mathcal{W}\, \ldots \,\mathcal{W}\, q_0)$; CHAIN prove response properties, of the form $\Box(p \rightarrow \Diamond\, q)$, that do not require the use of well-founded orders; and RANK diagrams prove response properties that do. Each type of verification diagram has different well-formedness constraints to ensure that the diagram satisfies the property.

We can define GVD templates for the more common properties, similarly to special-purpose diagrams. For example, to prove an invariance $\Box\, p$, the acceptance condition includes all SCS's, so no progress verification conditions need to be proved, and each node must be shown to satisfy p. The result is equivalent to the INVARIANCE diagrams of [MP94].

Example 5. Figure 4 provides a GVD template to prove formulas of the form $\Box(p \rightarrow \Diamond\, \Box\, q)$. The acceptance condition is

$$\mathcal{F} : \{\{n_0\}\, , \{n_2\}\} \ \ .$$

Thus all SCSs appearing in this circle must be shown to be transient: they should have a fair exit or be well-founded. The diagram of Figure 2 is an instance of this template. □

In general, the starting template for a GVD can be obtained from the *temporal tableau* of φ, a finite-state ω-automaton that represents all the models of φ (see, e.g. [MP95]).

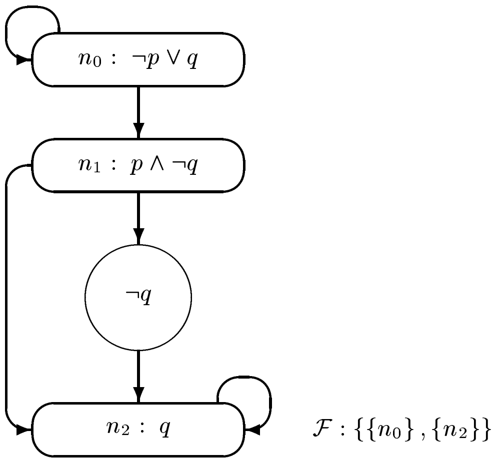

Fig. 4. GVD template to prove properties of the form $\Box(p \rightarrow \Diamond \Box q)$

In this case, the property satisfaction check is guaranteed to succeed, and the propositional labeling can be the identity. The user will have to prove consecution over this diagram, and show that all non-accepting SCS's have a fair exit or are well-founded. The structure of the diagram can then be filled in, adding details according to the user's understanding of the system. Verification diagrams can be constructed and checked incrementally. Since the verification conditions are local to the diagram, portions of the diagram can be formally verified, while others are edited until they can be proved correct or an error is found in the system being verified (see Section 6).

5 Related Work

While verification diagrams give a direct proof that all computations of a system satisfy the property φ, *deductive model checking* (DMC) [SUM98] shows that no computation can satisfy its negation. This is done by transforming a *falsification diagram*, which represents the product of a system abstraction and the tableau for $\neg\varphi$. The system abstraction is *refined*, as necessary, until the property is proved or a counterexample is found. At any given point, the falsification diagram includes all the computations of the system that may possibly violate φ.

While a GVD shows that every computation of $\mathcal{S}$ *can* follow an accepting path through the diagram, DMC shows that every computation of $\mathcal{S}$ *must* end in an unsuitable SCS in the product graph.

Property-preserving abstractions for reactive systems are discussed in, for instance, [CGL94,LGS+95,Dam96]. For a more extensive discussion of deductive-algorithmic verification and abstraction, see [Uri98]. See [Sip98] for more on deductive model checking, diagrams, and their application to the verification of real-time and hybrid systems.

6 Implementation: the STeP System

The Stanford Temporal Prover (STeP) is a tool for the temporal verification of reactive systems [BBC$^+$96,MBB$^+$98]. STeP parses SPL programs into fair transition systems, or can take transition systems directly as input. STeP includes verification rules and diagrams, automatic invariant generation, and symbolic and explicit-state LTL model checking.

The latest version of STeP, 2.0, features a Java graphical user interface that facilitates the construction and verification of diagrams, including GVD's (currently, using Streett acceptance conditions). The system automatically generates the required verification conditions and performs the required ω-automata model checking step. The diagram editor is closely integrated with the proof editor, so that the user can visualize the portions of the diagram that are proved and unproved as the diagram is incrementally constructed.

STeP provides integrated first-order automated reasoning and decision procedures [Bjø98] to facilitate the proof of verification conditions. STeP also uses these procedures for the generation of finite-state abstractions [CU98]. STeP can verify real-time and hybrid systems [BMSU97,MS98], and is being extended to include modular specification and verification [FMS98].

STeP is freely available for research and educational use. For more information, see the STeP web pages, at:

`http://www-step.stanford.edu`

References

[BBC$^+$96] N.S. Bjørner, A. Browne, E.S. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In R. Alur and T.A. Henzinger, editors, *Proc. 8th Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, pages 415–418. Springer-Verlag, July 1996.

[BCM$^+$92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[Bjø98] N.S. Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Computer Science Department, Stanford University, November 1998.

[BMS95] A. Browne, Z. Manna, and H.B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, vol. 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[BMS96] A. Browne, Z. Manna, and H.B. Sipma. Hierarchical verification using verification diagrams. In 2nd *Asian Computing Science Conf.*, vol. 1179 of *LNCS*, pages 276–286. Springer-Verlag, December 1996.

[BMSU97] N.S. Bjørner, Z. Manna, H.B. Sipma, and T.E. Uribe. Deductive verification of real-time systems using STeP. In *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CU98] M.A. Colón and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In A.J. Hu and M.Y. Vardi, editors, *Proc.* 10^{th} *Intl. Conference on Computer Aided Verification*, vol. 1427 of *LNCS*, pages 293–304. Springer-Verlag, July 1998.

[Dam96] D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, July 1996.

[FMS98] B. Finkbeiner, Z. Manna, and H.B. Sipma. Deductive verification of modular systems. In W.P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference, COMPOS'97*, vol. 1536 of *LNCS*, pages 239–275. Springer-Verlag, 1998.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comp. Prog.*, 8:231–274, 1987.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.

[MBB+98] Z. Manna, N.S. Bjørner, A. Browne, M. Colón, B. Finkbeiner, M. Pichora, H.B. Sipma, and T.E. Uribe. An update on STeP: Deductive-algorithmic verification of reactive systems. In *Tool Support for System Specification, Development and Verification*, pages 87–91. Christian-Albrechts-Universitat, Kiel, June 1998. Full version to appear in LNCS.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J.C. Mitchell, editors, *Proc. International Symposium on Theoretical Aspects of Computer Software*, vol. 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[MS98] Z. Manna and H.B. Sipma. Deductive verification of hybrid systems using STeP. In T. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, vol. 1386 of *LNCS*, pages 305–318. Springer-Verlag, 1998.

[Sip98] H.B. Sipma. *Diagram-based Verification of Discrete, Real-time and Hybrid Systems*. PhD thesis, Computer Science Department, Stanford University, December 1998.

[SUM98] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. To appear in *Formal Methods in System Design*, 1998. Preliminary version appeared in *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, Springer-Verlag, pp. 208–219, 1996.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, pages 133–191. Elsevier Science Publishers (North-Holland), 1990.

[Uri98] T.E. Uribe. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Computer Science Department, Stanford University, December 1998.