Formal Methods in System Design, ?, 1–45 (1999)

Verifying Temporal Properties of Reactive Systems: A STeP Tutorial *

NIKOLAJ S. BJØRNER, ANCA BROWNE, MICHAEL A. COLÓN, BERND FINKBEINER, ZOHAR MANNA, HENNY B. SIPMA AND TOMÁS E. URIBE

manna@cs.stanford.edu

Computer Science Department, Stanford University, Stanford, California 94305

Abstract. We review a number of formal verification techniques supported by STeP, the Stanford Temporal Prover, describing how the tool can be used to verify properties of several versions of the Bakery algorithm for mutual exclusion. We verify the classic two-process algorithm and simple variants, as well as an atomic parameterized version. The methods used include deductive verification rules, verification diagrams, automatic invariant generation, and finite-state model checking and abstraction.

Keywords: temporal logic, deductive verification, verification diagrams, model checking

1. Introduction

Reactive systems maintain an ongoing interaction with their environment, and their computations are infinite sequences of states. A large number of systems can be seen as reactive systems, including hardware, concurrent programs, network protocols, and embedded systems. Temporal logic provides a convenient language for expressing properties of reactive systems. A temporal verification methodology provides techniques for proving that a given system satisfies a given temporal property.

This paper describes a number of techniques for the temporal verification of reactive systems. We have implemented these methods as part of the Stanford Temporal Prover verification system, STeP. They are illustrated by verifying properties of different versions of the well-known Bakery algorithm for mutual exclusion. We use STeP to conduct these formal proofs, describing the main features of the verification system along the way. In doing so, we hope to provide an introduction to the main concepts and tools used.

1.1. Finite-state, infinite-state, and parameterized systems

The deductive framework of STeP allows the verification of a broad class of reactive systems, which we now briefly describe.

* This research was supported in part by the National Science Foundation under grant CCR-9804100, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, by the Army under grants 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).

We say that a system is *infinite-state* if its computations can reach infinitely many distinct states. Such systems contain variables that range over unbounded domains. Most software can be classified as infinite-state, since data structures such as integers, lists and trees are best thought of as unbounded. Hardware systems, on the other hand, are usually *finite-state*, since they can be in only finitely many distinct states, represented by a fixed number of bits. Note that computations of finite-state systems are still infinite sequences of states—it is the number of such distinct states that is finite. We will see that while *model checking* tools can often automatically verify properties of finite-state systems, deductive tools allow verifying infinite-state systems as well, with some user interaction.

A system with variables over unbounded domains will be classified as finite-state if only finitely many states are reachable in its computations. However, this may not be evident from the system specification and, as we will see, different techniques are applicable in these different cases.

Another class of systems to be verified is introduced by *parameterization*. A parameterized system has an arbitrary number of replicated components; for instance, nodes in a network protocol, or processors and buses in a multiprocessor architecture. Deductive formalisms provide a natural way of verifying the general correctness of parameterized systems, for an arbitrary number of components.

New dimensions of infinity are introduced when considering *real-time systems*, where time advances continuously and the time elapsed between events can be measured. A further extension is given by *hybrid systems*, where continuous variables evolve over time as determined by differential equations.

In this paper, we will focus on tools for untimed systems, including parameterized ones. However, our framework can be used as the basis for the verification of real-time and hybrid systems as well [34, 10, 51].

1.2. The 2-process Bakery algorithm

Figure 1 shows BAKERY(2), a program that implements Lamport's *Bakery algorithm* for mutual exclusion [37, 38]. The program is written in the Simple Programming Language (SPL) of [46, 50], which is accepted as input by STeP. Two processes, P1 and P2, coordinate access to a *critical section*, where at most one process should reside at any given time. Deciding which process enters the critical section is similar to serving customers at a bakery, where the processes are the customers and the baker is the critical section, which can serve only one customer at a time. Each process selects a "ticket number" in y_1 and y_2, as the customers in a store would, and the process with the lowest number is allowed to enter the critical section. This is an infinite-state program since the value of the integer variables y_1 and y_2 may grow arbitrarily large. A ticket with value 0 indicates that the process is not interested in accessing the critical section.

This informal description of the algorithm can be made precise if the system and its properties are modeled formally. The properties we would like to show are:

- *Mutual exclusion*: control is never at locations ℓ_3 and m_3 at the same time.

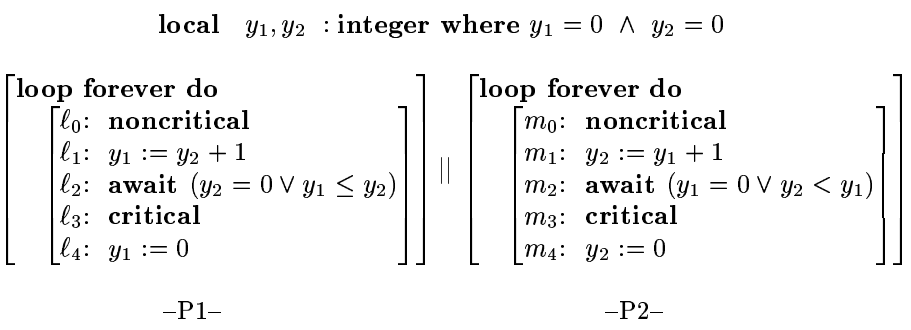

Figure 1. Program BAKERY(2)

- *One-bounded overtaking*: if one process wants to enter the critical section, the other process can enter the critical section at most once before the first one does.
- *Accessibility*: if control resides at ℓ_1 (resp. m_1), it will always eventually reach ℓ_3 (resp. m_3). That is, once a process has expressed interest in entering the critical section, it will eventually do so.

Note that it is not immediately clear, from observation of the program, which of these properties hold. In fact, early solutions to mutual exclusion suffered from *starvation*, where a process could forever be denied access to the critical section, and other solutions do not satisfy one-bounded overtaking. The Bakery algorithm of [37] was the first solution that satisfied the three main properties above for the general N-process case [50].

1.3. Outline

A formal verification framework requires three main components [47]:

1. A formal *computational model*, for describing systems.
2. A formal *specification language*, for describing properties of systems.
3. *Verification techniques* for establishing the validity of a property over a given system, such as a formal proof system or an algorithmic procedure.

By "formal," we mean that the methodology should be grounded on mathematical concepts that allow the specification of systems and properties in a precise, unambiguous way.

We describe our computational model, fair transition systems, in Sect. 2.1. Our property specification language, linear-time temporal logic, is presented in Sect. 2.2. The rest of this paper introduces our deductive and algorithmic verification techniques.

The STeP verification system is introduced in Sect. 3. In Sect. 4 we describe how *invariants* are automatically generated by STeP. In Sect. 5 we describe the use of *verification rules* for proving safety properties. In Sect. 6, we turn to *verification diagrams*, a visual representation of a proof that a reactive system satisfies a temporal specification.

An important technique is *abstraction*, where the system being verified is transformed into a simpler one, easier to verify, while preserving the properties of interest. We describe how this is done for BAKERY(2) in Sect. 7, where a simpler finite-state version is model checked.

Finally, we turn our attention to the general N-process (parameterized) algorithm in Sect. 8. We consider a simple parameterized version, and prove the three main properties for this program as well.

2. Preliminaries

2.1. Fair transition systems

To carry out formal verification, we need a precise mathematical model of the systems being analyzed. For this, we use *fair transition systems* [46] as our computational model for reactive systems. A fair transition system (FTS) $\mathcal{S}$ includes a set of variables $\mathcal{V}$, an *initial condition* Θ, and a finite set of *transitions* $\mathcal{T}$. A finite set of *system variables* $\vec{x} \subset \mathcal{V}$ determines the possible states of the system. The *state space*, Σ, is the set of all possible valuations of the system variables, each valuation defining a *system state*.

Fair transition systems can describe a wide range of systems, including hardware and software, using the full expressive power of first-order logic. We use a first-order assertion language $\mathcal{A}$ to describe the initial condition Θ and the transitions in $\mathcal{T}$. This assertion language can be augmented with interpreted function symbols and constraints, or specialized to the finite-state case, depending on the system being specified. An assertion over the system variables $\vec{x}$ describes the set of states for which it is true; if a state s satisfies the assertion φ, we say that s is a *φ-state*.

The initial condition Θ is such an assertion over $\vec{x}$, describing the set of *initial states*. A transition τ is described by a *transition relation* $\rho_\tau(\vec{x}, \vec{x}')$. This is an assertion over the set of system variables $\vec{x}$, indicating their values at a given state, and a set of *primed variables* $\vec{x}'$, which indicates their values at the next state. We assume that $\mathcal{T}$ always includes the *idling transition*, *Idle*, whose transition relation is $\rho_{Idle} : \vec{x} = \vec{x}'$.

A *run* is an infinite sequence of states $s_0, s_1, \ldots$ such that (1) s_0 satisfies Θ (*initiation*), and (2) for each $i \geq 0$, there is some transition $\tau \in \mathcal{T}$ such that $\rho_\tau(s_i, s_{i+1})$ is true (*consecution*); we say that τ is *taken* at s_i, and that state s_{i+1} is a *τ-successor* of s. A transition τ is *enabled* at a given state if it can be taken. Such states are characterized by the assertion

$$enabled(\tau) \stackrel{\text{def}}{=} \exists \vec{x}'.\rho_\tau(\vec{x}, \vec{x}') \ .$$

Fairness

Concurrency is modeled by *interleaving*: at any given point in a computation, a transition τ is executed, nondeterministically chosen from the set of transitions that are enabled at that point. A next-state that satisfies the transition relation for τ is selected, and the process repeated, building an infinite sequence of states that is a run of the system. This provides a simple encoding of non-determinism, and abstracts away the relative speed of different processes, since a sequence of transitions from one process can be taken before transitions for other processes are. However, we then want to exclude runs where a given process or non-deterministic action is forever ignored (the process never advances or an action never occurs even though it is allowed).

For this, the transitions in $\mathcal{T}$ can be marked as *just* or *compassionate*. A just (or *weakly fair*) transition cannot be *continually* enabled without ever being taken. A compassionate (or *strongly fair*) transition cannot be enabled infinitely often but taken only finitely many times. Given a transition system, we let $\mathcal{J}$ be the set of just transitions, and $\mathcal{C}$ be the set of compassionate ones. A *computation* is a run that satisfies these fairness requirements. Note that the presence of the idling transition ensures that all finite run prefixes can be extended to an infinite computation (even though there are no fairness requirements associated with *Idle*). A state is *reachable* if it appears in some computation of $\mathcal{S}$.

Example: Transition system for BAKERY**(2)**

STeP translates programs such as BAKERY(2) of Fig. 1 into the corresponding fair transition systems. For BAKERY(2), the state-space is given by the values of the system variables y_1, y_2, and two *control variables:* for each process, the fair transition system includes a variable that ranges over its distinct locations, indicating where control resides. STeP generates the control variables π_0 for P1 and π_1 for P2, each ranging from 0 to 4. Thus, the transition relation associated with the **await** program statement at ℓ_2 is:

$$\rho_{\ell_2} : \pi_0 = 2 \wedge (y_2 = 0 \vee y_1 \leq y_2) \wedge \pi_0' = 3 \wedge \pi_1' = \pi_1 \wedge y_1' = y_1 \wedge y_2' = y_2 \ .$$

The assertions ℓ_i and m_j are abbreviations for $\pi_0 = i$ and $\pi_1 = j$, indicating that control resides at locations i and j for P1 and P2. We also write $\ell_{2,3,4}$ to abbreviate the assertion $\pi_0 = 2 \vee \pi_0 = 3 \vee \pi_0 = 4$.

By the semantics given to the programming language constructs, all transitions of this program are just, except for the **noncritical** statements at m_0 and ℓ_0, which have no fairness requirements; they abstract away all process activity that is not related to the mutual exclusion protocol. Since these transitions are not fair, we cannot claim, for example, that if control is at ℓ_0 then it will eventually be at ℓ_1. Thus, the absence of a fairness requirement for ℓ_0 models the possibility that the statement may not terminate.

2.2. Linear-time temporal logic

Temporal logic is a convenient formalism for specifying and verifying properties of reactive systems, as first noted by Pnueli [56]. A formula of temporal logic describes the set of infinite sequences for which it is true, also known as a *temporal property*. A given system satisfies a property if all of its computations belong to this set.

A formula with no temporal operators is called a *state-formula* or an *assertion*. For an assertion f and state s, we say that f holds at s if f is true given the values of the system variables at s, and we say that s is an *f-state*.

A model of *linear-time temporal logic* (LTL) is an infinite sequence of states. Given a model $\sigma : s_0, s_1, \ldots$, the *future* temporal operators are defined as follows:

We say that a temporal formula φ holds at position s_i of a model σ if φ is true for the sequence $s_i, s_{i+1}, \ldots$ that starts at that position, and write $(\sigma, i) \models \varphi$. If f is an assertion, then $(\sigma, i) \models f$ iff f holds at state s_i. Then:

$(\sigma, i) \models p \wedge q$ iff $(\sigma, i) \models p$ and $(\sigma, i) \models q$;

$(\sigma, i) \models p \vee q$ iff $(\sigma, i) \models p$ or $(\sigma, i) \models q$;

$(\sigma, i) \models \neg p$ iff $(\sigma, i) \not\models p$;

$(\sigma, j) \models \Box\, p$ iff $(\sigma, i) \models p$ for all $i \geq j$;

$(\sigma, j) \models \Diamond\, p$ iff $(\sigma, i) \models p$ for some $i \geq j$;

$(\sigma, j) \models \bigcirc\, p$ iff $(\sigma, j+1) \models p$;

$(\sigma, j) \models p\, \mathcal{U}\, q$ iff $(\sigma, k) \models q$ for some $k \geq j$, and $(\sigma, i) \models p$ for every i, $j \leq i < k$.

$(\sigma, j) \models p\, \mathcal{W}\, q$ iff $(\sigma, j) \models (p\, \mathcal{U}\, q)$ or $(\sigma, j) \models \Box\, p$.

The model σ satisfies φ, written $\sigma \models \varphi$, if $(\sigma, 0) \models \varphi$.

Past versions of these temporal operators are similarly defined, but in this paper we will only use the future fragment of LTL. Note that we do not allow temporal operators to appear within the scope of a quantifier, so all quantification can be thought of as *rigid*, independent of time. (System variables are *flexible* variables, since their values can change over time.) As in [50], formulas that satisfy this restriction are called *state-quantified* formulas. The STeP verification methods are only applicable to these formulas (even though the theorem-proving component is not restricted to state-quantified formulas, but also handles formulas with quantifiers outside temporal operators). A formula of the form $\Box\, f$ for an assertion f is called an *invariance* formula, or *invariant*.

Given a system $\mathcal{S}$, we say that a temporal formula φ is *$\mathcal{S}$-valid* if every computation of $\mathcal{S}$ satisfies φ, and write $\mathcal{S} \models \varphi$. We say that an assertion p is *$\mathcal{S}$-state valid* if p holds on all states of all computations of $\mathcal{S}$, that is, if it holds on all reachable states.

Example: Mutual exclusion for the BAKERY(2) program of Fig. 1 can be expressed with the invariance formula

$$\Box \neg(\ell_3 \wedge m_3) \ .$$

One-bounded overtaking is expressed by the *nested wait-for formula*

$$\Box(\ell_2 \ \rightarrow \ \neg m_3 \,\mathcal{W}\, (m_3 \,\mathcal{W}\, (\neg m_3 \,\mathcal{W}\, \ell_3))) \ .$$

This formula states that whenever control is at ℓ_2, meaning that P1 wants to enter the critical section (ℓ_3), the following must occur: there may be an interval in which P2 is not in the critical section (so all states in the interval satisfy $\neg m_3$), followed by an interval where P2 is in the critical section (states satisfying m_3), followed by an interval where P2 is again not in the critical section (states satisfying $\neg m_3$), followed finally by a state where P1 is in the critical section (ℓ_3). Thus, P2 can enter the critical section (m_3) *at most once* before P1 enters (ℓ_3). Each of the intervals $\neg m_3$, m_3, and $\neg m_3$ that precede ℓ_3 can possibly be empty but (as far as this formula is concerned) could extend indefinitely as well.

Accessibility ensures that these intervals do not extend indefinitely, and is expressed with the *response formula*

$$\Box(\ell_1 \rightarrow \Diamond\, \ell_3) \ .$$

That is, it is always the case that if control is at ℓ_1, then it will eventually reach ℓ_3.

We often write $p \Rightarrow q$ as an abbreviation for $\Box(p \rightarrow q)$. Thus, accessibility can be written as $\ell_1 \Rightarrow \Diamond\, \ell_3$, and one-bounded overtaking as

$$\ell_2 \ \Rightarrow \ \neg m_3 \,\mathcal{W}\, m_3 \,\mathcal{W}\, \neg m_3 \,\mathcal{W}\, \ell_3$$

where the wait-for operator $\mathcal{W}$ associates to the right.

Mutual exclusion and one-bounded overtaking belong to the class of temporal *safety* properties. Informally, such properties state that something "bad" can never happen, and are falsified if a bad state is ever reached: if a safety formula φ is false in a model, then there is a finite prefix of the model such that φ is also false in every extension of this prefix.

Accessibility, on the other hand, is a *response* property, and belongs to the larger class of *progress* properties. These properties state that something "good" is guaranteed to happen eventually. As we will see in Sect. 3.7, the verification of each class of properties has its particular requirements. For instance, safety properties are independent of the fairness requirements of the given transition system, whereas the verification of response properties relies on fairness.

A detailed exposition of linear-time temporal logic as a specification language for fair transition systems is found in [46]. See [18] for formal definitions of safety and progress classes.

Finally, we define *ranking functions*, which we will use to express well-founded induction arguments in proofs: A binary relation $\succ$ is *well-founded* over a domain $\mathcal{D}$ if there is no infinite descending chain, that is, no infinite sequence of elements $e_1, e_2, \ldots$ in $\mathcal{D}$ such that $e_i \succ e_{i+1}$ for all $i \geq 0$. 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.

2.3. Verification conditions

We will use *verification rules* and *verification diagrams* to reduce the validity of a temporal property over a given system to the general validity of first-order *verification conditions*. While temporal formulas express global properties of the system, which should hold over an entire computation, verification conditions express more local properties, describing how individual transitions may lead from one set of states to another in a single step.

In particular, we will use verification conditions to state that a transition τ, if executed in a state that satisfies φ, is guaranteed to reach a state that satisfies ψ (if it can be executed at all), for assertions φ and ψ. We use a special notation for this, known as the *Hoare triple*:

$$\{\varphi\}\,\tau\,\{\psi\} \quad \stackrel{\text{def}}{=} \quad \forall \vec{x}.\forall \vec{x}'.\,(\varphi(\vec{x}) \wedge \rho_\tau(\vec{x}, \vec{x}')) \rightarrow \psi(\vec{x}') \ .$$

Note that this verification condition holds if τ is not enabled at any φ-state.

3. The STeP system

The Stanford Temporal Prover, STeP, is a tool for the formal verification of temporal properties of reactive systems [6, 7]. STeP implements verification rules and verification diagrams, provides automated support for proving verification conditions, and allows model checking whenever possible.

Figure 2 presents an overview of STeP. The top of the diagram indicates the main inputs accepted by STeP. System descriptions are expressed in some variant of the fair transition system model, where *clocked* and *phase* transition systems are extensions suitable for real-time and hybrid systems respectively. Temporal properties are expressed as LTL formulas.

Verification rules or model checking are used to prove temporal properties. Automatic theorem-proving and invariant generation mechanisms are available to facilitate this task. The user provides proof guidance in two main forms: user-provided verification diagrams give a high-level, yet formal proof of the system validity of a particular property. Interactive theorem-proving can establish, with user guidance, verification conditions that the (incomplete) automatic prover cannot prove.

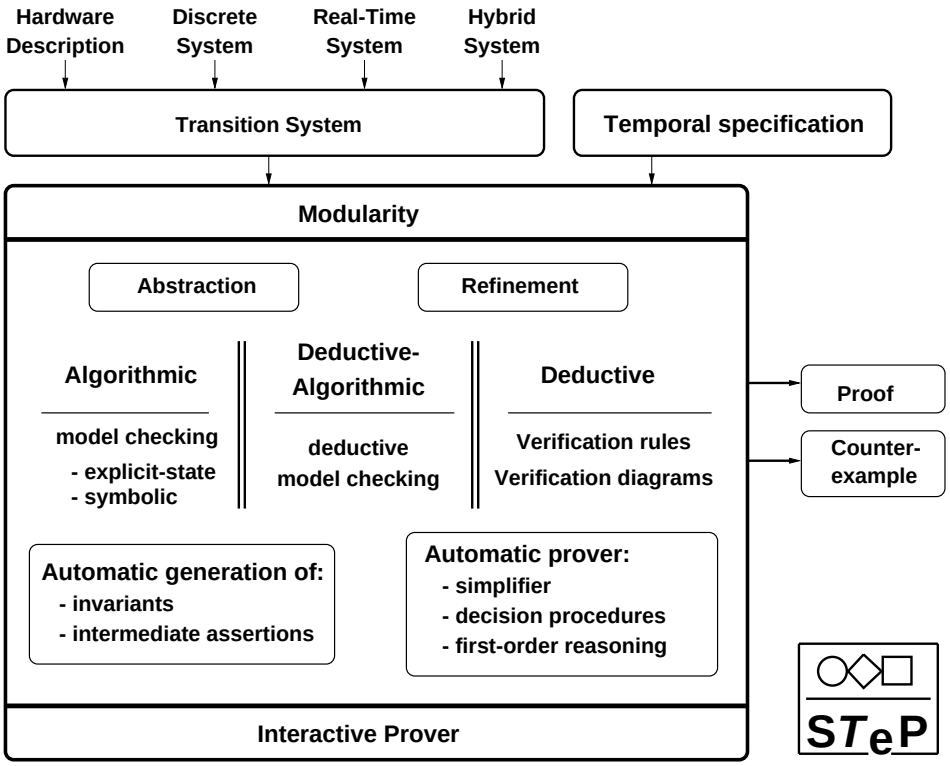

Figure 2. An overview of the STeP system

3.1. Verification methodology

STeP is best viewed as providing a toolkit of verification methods based on a common system description language and specification language. A given system can be analyzed in a number of ways. Depending on the system and property to be proved, different tools will be applicable or more appropriate:

Model Checking: If the system is finite-state, arbitrary temporal properties can be automatically established or refuted, using explicit-state or symbolic model checking (Sect. 3.6). (Symbolic model checking is applicable only if all variables have finite-domain. Some classes of infinite-state systems can be checked with the explicit-state model checker, which is not guaranteed to terminate in this case.) For parameterized and infinite-state systems, finite-state instances of the system can be model checked to search for bugs.

Invariant Generation: For most deductive verification proofs, system invariants of increasing strength must be collected, where previous invariants are used to

establish subsequent ones. Automatic invariant generation is used to establish a basic initial set of invariants (Sect. 4).

Verification Rules: To prove simple safety properties, deductive rules can be used, with the user providing adequate strengthenings (intermediate assertions) where necessary (Sect. 5). Previously established invariants are used to prove the required verification conditions, which are automatically generated by the system.

Verification Diagrams: A *verification diagram* can be provided by the user, as a system abstraction that proves a particular property in question. Verification conditions justify the correctness of the diagram, while an algorithmic procedure checks that the diagram, indeed, proves the property at hand (Sect. 6).

Abstraction: A *finite-state abstraction* of the system can be generated, such that properties model checked for the abstract system will hold of the original system as well (Sect. 7). Since the abstraction is finite-state, it can be model checked. Available invariants improve the quality of the abstraction, allowing more properties to be proved.

3.2. User interaction

STeP provides a graphical user interface, implemented in Java, in the form of three editors: a *session editor*; a *proof editor*; and a *verification diagram editor*. We now briefly describe the functionality and interaction that each provides. Subsequent sections describe the underlying formalisms in more detail.

The Session Editor: STeP's user interface is designed to support the notion of a *verification session*, which is a collection of related verification tasks. The session editor allows the user to manipulate these sessions and serves as the principal interface to the verification methods implemented in STeP. The session editor is used to load systems and their specifications, and it handles the verification of multiple related systems. It is used to invoke the automatic invariant generation and abstraction algorithms on selected systems and provides direct access to the model checker for those systems which are finite-state. It also offers facilities to save and load entire sessions, making it possible to postpone a verification effort and continue with it at a later time.

The session editor maintains the database of properties to be proven, and allows the user to specify which of these properties are considered *active*, that is, useable as lemmas when proving or model checking other properties. By default, the axioms of specifications and the generated invariants are considered active, but they may be selectively deactivated to improve the performance of the simplifier and the validity checker. In addition, the session editor maintains a graph relating properties to the assumptions used in their proofs. By continuously updating this graph, the session editor offers the flexibility of assuming unproven conjectures when proving properties. Soundness is maintained by prohibiting circular reasoning.

The Proof Editor: Proofs in STeP are maintained as trees in which each node is labeled by a property or a sequent, where sequents are manipulated by a number

of Gentzen-style rules for verification conditions which cannot be established automatically. In addition, nodes are marked as either *open* or *closed*, and a proof is considered complete when all of its nodes are closed. Verification rules with premises close nodes by adding one open child for each premise, while premise-free rules close nodes without expanding the tree.

The proof editor provides facilities for viewing and manipulating proofs. For easy navigation through the proof, it presents a graphical view of the entire tree, along with an iconic indication of the status of each node. The most frequently used verification rules are accessed through a toolbar, while the rest are relegated to menus. Special rules are available for converting between property and sequent forms of verification conditions, and the toolbar switches automatically between standard and Gentzen rules based on the label of the current node.

The proof editor also supports a powerful mechanism for re-running proofs. Closed nodes in a proof are marked as either *valid* or *invalid*. When a node is first closed by a verification rule, it is considered valid. If the user later takes an action which renders that application of the rule suspect, the node is marked as invalid, but the subproof rooted at the node is preserved. For example, if the user changes the assertion labeling a node in a verification diagram, all closed proof nodes corresponding to its consecution conditions are invalidated. The user can later direct the proof editor to revalidate the proof, which entails visiting all invalid nodes, re-applying their rules, and relating the old and new children in an attempt to preserve as much of the structure of the proof as possible.

The Verification Diagram Editor: The third main component of the STeP user interface is the verification diagram editor. It allows the user to construct verification diagrams, which can then be used as specialized verification rules in the proof editor. This graphical editor supports the direct manipulation of diagrams and provides features to ease the incremental construction of diagrams. The editor can provide the user with an initial template of a diagram, which is derived from the temporal *tableau* of the formula the diagram is intended to establish. The editor supports the use of multiple diagrams in a single proof, allowing decomposition of the verification task into smaller diagrams. Each individual diagram proves an aspect of the property to prove, or proves an auxiliary property.

We describe verification diagrams in Sect. 3.7.

3.3. Decision procedures and validity checking

The verification conditions generated by verification rules and verification diagrams are first-order formulas, which should be proved valid, perhaps with respect to a set of axioms and previously established invariants (which we call *background properties*).

Rather than general first-order validity, the validity of verification conditions is relative to particular theories relevant to the domains of computation. This includes, for example, the theory of arithmetic. Rather than axiomatize all of these theories, a more convenient approach is to use *decision procedures* that can automatically establish the validity of formulas in some of these theories.

STeP provides a number of decision procedures and rewriting mechanisms, which are combined in a *simplifier* [42]. Verification conditions that are simplified to *true* are automatically discharged. For instance,

$$(y \geq 0 \ \wedge \ y' = y + 1) \ \rightarrow \ y' > 0$$

will be simplified to *true* using STeP's decision procedures for linear arithmetic. Decision procedures for equality and uninterpreted symbols can automatically establish, for example, the validity of

$$a = f(a) \ \wedge \ a = b \ \rightarrow \ f(f(f(b))) = a \ .$$

Finally, the decision procedures for the various theories are combined within a Shostak-style integration [59] to establish formulas such as

$$a > b \ \wedge \ b + 1 \geq a \ \rightarrow \ f(a + b) = f(2b + 1)$$

where a and b are integers.

The simplifier is designed to be fast rather than complete. The strength of the simplifier is determined by a number of user-controlled *simplification flags*. These determine, for instance, whether decision procedures for linear-arithmetic, matching for associative-commutative function symbols, or conditional rewrite rules are applied.

Validity checking: Decision procedures usually operate on *ground formulas*, which contain no quantifiers. We have developed a combination of ground-level decision procedures and first-order quantifiers intended to establish the validity of verification conditions which require "trivial" quantifier instantiations.

The `Check-Valid` utility in STeP checks the validity of first-order verification conditions relative to background axioms. It provides a complete decision procedure for a large decidable fragment of ground theories as well as a bounded search for quantifier instantiations [11]. Unlike the simplifier, the goal of this procedure is to prove the formula's validity, rather than rewrite it to an equivalent form. If the validity check fails, the formula remains unchanged. The theories integrated in STeP's decision procedures include equality (congruence closure), partial orders, arrays, records, linear and non-linear arithmetic over the reals, bit-vectors, lists, queues, well-founded sets, and datatypes for finite and infinite trees. Each procedure solves only constraints in its domain in isolation, and a tight integration within congruence closure eliminates redundant constraint propagation. The specifics of the decision procedure integration are described in more depth in [5].

3.4. Interactive theorem-proving

First-order validity is undecidable. For those verification conditions that are not proved valid by the automatic tools, the interactive prover can be used. This prover is complete for (uninterpreted) first-order logic: if the formula is valid, a proof for

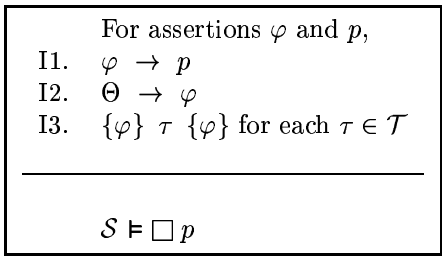

Figure 3. General invariance rule G-INV

it can be found. (This prover can be used to reason about general LTL properties as well, including flexible and rigid quantification, but such a system cannot be complete in general.)

The Gentzen prover builds a *proof tree*. Each node of the tree is associated with a subgoal, where the root is the formula to be proved. At each step, a rule is applied, which reduces the validity of the current node to the validity of its children. At the leaves, the validity of the subgoal is established using the automatic decision procedures and validity checking procedures.

User-guidance comes in three main forms: the choice of rule to apply, such as induction over naturals, integers, and well-founded data-types; the use of *cut formulas*, which are arbitrary user-provided formulas used to perform case analysis; and the duplication and instantiation of quantified formulas.

When the Proof Editor is in Gentzen mode, the simplifier does not use background properties, unless they are explicitly added by the user, giving the user more control over the proofs.

3.5. Deductive verification

The classical deductive framework for the verification of fair transition systems is based on *verification rules*, which reduce temporal properties of systems to first-order premises.

Figure 3 presents the *general invariance rule*, G-INV, which can be used to prove the $\mathcal{S}$-validity of formulas of the form $\Box p$ for an assertion p. The premises of the rule are first-order verification conditions. If they can be proved to be valid, the temporal conclusion must hold for the system $\mathcal{S}$.

We say that an assertion is *inductive* if it is preserved by all the transitions of a system and holds at all initial states. The invariance rule relies on finding an inductive auxiliary assertion φ that strengthens p; the invariant p may not be inductive, but φ must be. Note that it may be possible for individual transitions to violate p (see Sect. 5.2). However, the rule ensures that the states where this happens do not appear in any run of $\mathcal{S}$.

The soundness of the rule is obvious: if φ holds initially and is preserved by all transitions, it will hold for every reachable state of $\mathcal{S}$. If p is implied by φ, then p will also hold for all reachable states.

Rule G-INV is also *relatively complete*: if p is an invariant of $\mathcal{S}$, then the strengthened assertion φ always exists, provided the assertion language is sufficiently expressive [45]. However, in the case of general infinite-state systems, proving invariants is undecidable, and finding such a φ can be nontrivial, requiring the use of auxiliary system variables or past temporal formulas in general.

Other verification rules can be used to verify different classes of temporal formulas, ranging from safety to progress properties [45]. These deductive methods are relatively complete and yield a direct proof of any valid property, but may require substantial user guidance to succeed and do not produce counterexample computations when the property fails.

Graphical formalisms can facilitate the task of guiding and understanding a deductive proof. *Verification diagrams* [48, 16] provide a graphical representation of the verification conditions needed to establish a particular temporal formula. We will see examples of specialized verification diagrams in Sects. 6.1 (*invariance diagrams*), 6.2 (*wait-for diagrams*) and 6.4 (*chain diagrams*). The generalized case is presented in Sect. 6.6.

3.6. Model checking

Model checking algorithms [19, 58] automatically determine the validity of temporal properties for finite-state systems. The classic model checking algorithms, designed for *branching-time* temporal logic, recursively label system states with the formulas they satisfy, starting with atomic propositions occurring in the formula and analyzing more complex subformulas at each step.

Automata-theoretic model checking: In the automata-theoretic approach to model checking [35, 36, 63], the system $\mathcal{S}$ can be identified with an ω-automaton that recognizes a language $\mathcal{L}(\mathcal{S})$ of infinite words, which is the set of all $\mathcal{S}$-computations. Similarly, the property to be verified can be expressed as an automaton Φ_φ, where the language $\mathcal{L}(\Phi_\varphi) = \mathcal{L}(\varphi)$, the set of all infinite sequences for which φ holds. To prove that $\mathcal{S}$ satisfies φ, we need to show that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Phi_\varphi)$. If both Φ_φ and $\mathcal{S}$ are finite automata, this language inclusion problem can be decided by purely algorithmic means.

STeP provides an explicit-state model checker that can automatically establish LTL properties of finite-state systems, provided the state-space is not too large. If the property does not hold, the model checker returns a counterexample computation. It can sometimes model check infinite-state systems as well, when only a finite portion of that space needs to be explored (but it is not guaranteed to terminate in this case) [42].

Symbolic model checking: For a finite-state system $\mathcal{S}$, the complexity of model checking depends on the size of the formula being checked (linear for the branching-time logic CTL and exponential for LTL) and the size of the state-space of $\mathcal{S}$ (linear).

While the temporal formulas of interest are usually small, the size of the state-space can grow exponentially in the size of its description, say, as a fair transition system (i.e. the number of processes). This is known as the *state explosion problem*, where the number of possible states limits the practical applicability of model checking algorithms.

Symbolic model checking [53] fights the state-explosion problem by using a specialized data structure to describe sets of states, rather than explicitly enumerating them. Ordered Binary Decision Diagrams (OBDDs) [17] are an efficient data structure for representing boolean functions and relations. OBDDs can represent the transition relation of finite-state systems, as well as subsets of the system's state-space. The efficient algorithms for manipulating OBDD's can be used to compute predicate transformations, such as pre- and post-conditions, over the transition relation and large, implicitly represented sets of states.

STeP includes a symbolic LTL model checker, applicable only to systems whose variables range over finite domains. It can be used to check the general validity of LTL formulas as well as to check properties over fair transition systems. It does not require an explicit representation of a temporal tableau, but uses a fixpoint computation to obtain an OBDD representing the set of reachable states that satisfy the negation of the property to be checked. The property being checked holds if and only if this fixpoint is the constant *false* OBDD.

3.7. Program BAKERY[2] *in STeP*

In Fig. 1 we presented BAKERY(2) written as an SPL program. This program can be entered into STeP as shown in Fig. 4.

The STeP parser translates this program into a fair transition system containing 11 transitions, one for each of the statements `l0..l4`, `m0..m4`, and the idling transition. This transition system will be the basis for the verification. As described in Sect. 2.1, two control variables are used for each of the processes; in STeP, these are `pi0` and `pi1`; the assertions `l0`, ..., `l4` (rather than $\ell_0, \ldots, \ell_4$) and `m0`, ..., `m4` are recognized by STeP as well, abbreviating `pi0=0`,...,`pi0=4` and `pi1=0`,...,pi1=4.

4. Automatic invariant generation for BAKERY[2]

Verification is usually an incremental process: simple properties of the system being verified are proved first, and then used to help establish more complex ones. For instance, when applying a verification rule such as G-INV of Fig. 3, the verification conditions in the premise can be proved valid relative to previously known invariants, which can be used as lemmas in the proof. That is, the verification conditions can be shown to be $\mathcal{S}$*-state valid*, i.e. true at every reachable state of $\mathcal{S}$, rather than generally valid. Invariants can also be used to constrain the state-exploration carried out in model checking.

It is therefore desirable to establish simple invariants of the system being analyzed. This can be carried out by automatic tools, from a static analysis of the system. A framework for the automatic generation of invariants is presented in [8], based on

```
local y1, y2 :int where y1 = 0, y2 = 0
P1::[loop forever do [
        l0: noncritical;
        l1: y1 := y2 + 1;
        l2: await (y2 = 0 \/ y1 <= y2);
        l3: critical;
        l4: y1 := 0
        ]
]
||
P2::[loop forever do [
        m0: noncritical;
        m1: y2 := y1 + 1;
        m2: await (y1 = 0 \/ y2 < y1);
        m3: critical;
        m4: y2 := 0
        ]
]
```

Figure 4. Two-process Bakery as entered in STeP

abstract interpretation techniques [22]. These techniques carry out an approximate propagation through the state-space of the system. Any assertion that describes a superset of the reachable states of the systems is a valid invariant. Different approximation techniques yield different invariants of varying strength.

Early work on generating invariants is [27, 23]; another recent approach is reported in [4].

The invariants described above are sometimes called *bottom-up* invariants, since they are generated from the system alone, independently of any property being verified. Techniques for finding *top-down* invariants, which are strengthened assertions that depend on the particular formula being verified, are also presented in [8]. We discuss these in more detail in Sect. 6.3.

Once the BAKERY(2) program is loaded, STeP can automatically generate invariants of the following kinds:

Local invariants: These invariants relate control and data with implications of the form "whenever control resides at location x then the values of the data variables are y." STeP automatically generates these local invariants for BAKERY(2):

$$\Box(y_1 \geq 0)$$
$$\Box(y_2 \geq 0)$$
$$\ell_{0,1} \Rightarrow y_1 = 0$$
$$m_{0,1} \Rightarrow y_2 = 0$$

In general, the set of states that are either initial or reachable from any state in one step of some non-idling transition is clearly a superset of the reachable states, and thus an invariant of the system. This set is characterized by the formula

$$\Theta \vee \bigvee_{\tau \in \mathcal{T} - \{Idle\}} \exists \vec{x}_0 \,.\, \rho_\tau(\vec{x}_0, \vec{x}) \,.$$

STeP eliminates the existentially quantified variables $\vec{x}_0$, solving for $\vec{x}_0$ whenever possible and replacing unsolvable conjuncts in ρ_τ by *true* (*weakening*). In this way, STeP generates the following invariant for BAKERY(2) (after simplification, where $\wedge$ has precedence over $\vee$):

$$\Box \begin{pmatrix} \ell_{1,4} \;\vee\; \ell_0 \wedge y_1 = 0 \;\vee\; \ell_2 \wedge y_1 = y_2 + 1 \\ \vee\; \ell_3 \wedge (y_1 \leq y_2 \vee y_2 = 0) \;\vee\; m_3 \wedge (y_2 < y_1 \vee y_1 = 0) \\ \vee\; m_{1,4} \;\vee\; m_0 \wedge y_2 = 0 \;\vee\; m_2 \wedge y_2 = y_1 + 1 \end{pmatrix} \tag{1}$$

These are sometimes called *reaffirmed invariants*.

Linear invariants: This class of invariants captures relationships between variables that can be expressed as linear equations. For BAKERY(2), STeP's linear invariant mechanism generates the obvious control invariants, stating that control will always be at exactly one of $\ell_0, \ldots, \ell_4$ and $m_0, \ldots, m_4$:

$$\Box \;\; 1 = (\ell_0 + \ell_1 + \ell_2 + \ell_3 + \ell_4)$$
$$\Box \;\; 1 = (m_0 + m_1 + m_2 + m_3 + m_4)$$

These formulas use *arithmetization*, where the value of a boolean expression is treated as 0 (*false*) or 1 (*true*) within an arithmetic expression. These two invariants are not needed in the forthcoming verification, and can be deactivated or removed from the list of background properties.

Polyhedral invariants: Finally, polyhedral invariants generalize linear invariants, capturing relationships between variables that can be expressed as linear equalities and inequalities. The polyhedral invariant generation tools in STeP add the following two invariants for BAKERY(2):

$$\ell_{2,3,4} \;\Rightarrow\; y_1 > 0$$
$$m_{2,3,4} \;\Rightarrow\; y_2 > 0$$

Here, we used STeP's default polyhedral invariant settings; these can be changed to generate stronger invariants (which will take longer).

All the above are bottom-up invariants, since they are generated from the analysis of the system independently of any temporal property to be proved. Once generated, these invariants are part of the set of *background properties*, which can also include axioms and previously proved properties. The `Simplify` proof rule uses the active background invariants as lemmas to simplify the given subgoal. Similarly, the `Check-Valid` rule uses the background properties to try to establish the validity of the goal.

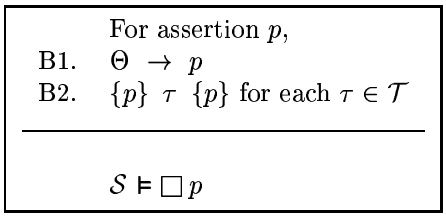

Figure 5. Basic invariance rule B-INV.

5. Verification rules

Verification rules are the preferred method in STeP for proving simple properties over infinite-state systems.

5.1. Mutual exclusion: rule B-INV

Mutual exclusion, expressed by

$$mux : \Box \neg(\ell_3 \wedge m_3) \ ,$$

is entered as a specification in STeP as

```
[]~(l3 /\ m3)
```

The first choice for verifying such invariance properties is the *basic invariance rule*, B-INV, shown in Fig. 5. This is a special case of the general invariance rule G-INV of Fig. 3, where $\varphi \equiv p$.

For BAKERY(2), rule B-INV reduces the proof of mux to the first-order validity of 12 verification conditions: one for each transition (including *Idle*), plus the implication for the initial condition. Consider now the verification condition for transition ℓ_2:

$$\neg(\ell_3 \wedge m_3) \ \wedge \ \rho_{\ell_2} \ \rightarrow \ \neg(\ell_3{}' \wedge m_3{}')$$

that is,

$$\neg(\ell_3 \wedge m_3) \ \wedge \ \underbrace{(y_2 = 0 \vee y_1 \leq y_2) \wedge \ell_2 \wedge \ell_3{}' \wedge \pi_1' = \pi_1 \wedge \ldots}_{\rho_{\ell_2}} \ \rightarrow \ \neg(\ell_3{}' \wedge m_3{}') \tag{2}$$

This verification condition is valid relative to the previously generated invariants. Using these invariants as background properties, the STeP validity checker proves this verification condition automatically.

5.2. Mutual exclusion: strengthening by backwards propagation

In the following, we will disregard the reaffirmed local invariant for BAKERY(2), in order to illustrate different methods for verifying invariants that cannot be established directly using B-INV. In such cases, the invariant we want to prove is not strong enough, and is not preserved by all the transitions; we say it is not *inductive*. There are several ways to proceed.

One solution is to find a stronger assertion that implies the proposed invariance, and is inductive. This approach is reflected in the more general rule G-INV, discussed in Sect. 3.5. Even though the strengthened assertion always exists if the invariant is P-valid, in general it may be hard to find. An alternative, more automatic approach is to perform *backwards propagation*, also called *assertion propagation*. Consider again the verification condition for ℓ_2 above (formula 2). The fact that it is not valid by itself indicates that there exist states where it is possible to take transition ℓ_2 and reach a state where mutual exclusion is violated. An example of such a state is one that satisfies

$$\ell_2 \wedge m_3 \wedge y_1 \leq y_2 \quad .$$

If mutual exclusion holds, this state cannot be reachable.

This is the main idea behind backwards propagation: all states from which a violating state is reachable by some transition must themselves be unreachable. To express this, we define the *weakest precondition* of an assertion φ with respect to a transition τ:

$$wlpc(\tau, \varphi) \stackrel{\text{def}}{=} \forall \vec{x}'. (\rho_\tau(\vec{x}, \vec{x}') \rightarrow \varphi(\vec{x}')) \quad .$$

This assertion describes the states that must reach a φ-state after τ is taken. This is sometimes called the *weakest liberal precondition*, since it is true for those states where τ cannot be taken at all. It is not difficult to see that if φ is an invariant, then so is $wlpc(\tau, \varphi)$: otherwise, there would be a reachable state where τ could be taken, leading to a state that violates φ. Thus we can always *strengthen* a property φ into

$$\varphi \wedge wlpc(\tau, \varphi)$$

for every transition τ such that $\{\varphi\}\,\tau\,\{\varphi\}$ is not P-state valid, and try to prove the resulting invariance instead.

However, in practice this leads to large formulas. An alternative approach is to prove separately that $wlpc(\tau, \varphi)$ is an invariant, and then use this to infer the validity of the verification condition that could not be proven. This strategy is strictly weaker than the strengthening approach, since it may be the case that neither φ nor $wlpc(\tau, \varphi)$ are inductive but that $\varphi \wedge wlpc(\tau, \varphi)$ is inductive. Both strategies are available as verification rules in STeP, called `Strengthen` and `WLPC`, respectively.

For our BAKERY(2) program, applying B-INV to prove mutual exclusion (without the reaffirmed invariant, but with the local, linear and polyhedral invariants of

For assertions $p, q_0, q_1, \ldots, q_m$, and $\varphi_0, \ldots, \varphi_m$,

N1. $p \rightarrow \bigvee_{j=0}^{m} \varphi_j$

N2. $\varphi_i \rightarrow q_i$, for $i = 0, \ldots, m$

N3. $\{\varphi_i\}\ \tau\ \{\bigvee_{j \leq i} \varphi_j\}$, for each $i = 1, \ldots, m$ and $\tau \in \mathcal{T}$

$\mathcal{S} \models p \Rightarrow q_m \mathcal{W} q_{m-1} \ldots q_1 \mathcal{W} q_0$

Figure 6. General nested wait-for rule G-WAIT

Sect. 4) results in two verification conditions that are not valid, for ℓ_2 and the symmetric case m_2. We can apply `WLPC` to these two verification conditions and obtain the two new proof obligations

$$\Box\left((y_1 = 0 \vee y_2 < y_1) \wedge m_2 \rightarrow \neg \ell_3\right)$$
$$\Box\left((y_2 = 0 \vee y_1 \leq y_2) \wedge \ell_2 \rightarrow \neg m_3\right) \quad .$$

Unlike *mux*, these invariants are inductive relative to the currently active invariants, and can thus be proven with rule B-INV. All the verification conditions are proved automatically, which completes the proof.

5.3. Bounded overtaking: rule G-WAIT

Once we have proved mutual exclusion we may wish to establish other desirable properties such as one-bounded overtaking, i.e. if process P1 is interested in entering the critical section, then process P2 cannot enter more than once before P1 does. As discussed in Sect. 2.2, this property is specified by:

$$\textit{one-bounded:}\ \ell_2 \ \Rightarrow\ \neg m_3 \mathcal{W} m_3 \mathcal{W} \neg m_3 \mathcal{W} \ell_3$$

or in STeP format:

```
l2 ==> ~m3 Awaits m3 Awaits ~m3 Awaits l3
```

The verification rule G-WAIT, shown in Fig. 6, reduces the P-validity of such *nested wait-for* formulas to a set of first-order verification conditions.

Similarly to the general invariance rule, G-WAIT offers the possibility of strengthening the intermediate assertions, going from q_i to φ_i, so that the verification conditions in premise N3 are P-state valid. Again, in general it is nontrivial to find these assertions, although the completeness of the rule guarantees that they always exist if the wait-for property is P-valid.

In our case, it is not too difficult to find the required strengthenings. Looking at the formula

$$\ell_2 \;\Rightarrow\; \underbrace{\neg m_3}_{q_3} \,\mathcal{W}\, \underbrace{m_3}_{q_2} \,\mathcal{W}\, \underbrace{\neg m_3}_{q_1} \,\mathcal{W}\, \underbrace{\ell_3}_{q_0}$$

we observe that φ_1 should represent all states where P1 has priority over P2 to enter the critical section. process P1 has priority if (a) P2 is not interested, i.e. P2 is at m_0 or at m_1 (and thus, using the local invariants, $y_2 = 0$), or (b) P2 has just left the critical section, i.e. m_4, or (c) P2 is interested, but has a higher ticket, i.e. P2 is at m_2 and $y_1 \leq y_2$. On the other hand, φ_3 need not be strengthened; it can cover both the case that P1 has priority over P2 and the case that P2 has priority over P1. Thus, the following strengthenings are suggested:

$$\begin{aligned}
\varphi_3 &: \neg m_3 \\
\varphi_2 &: \ell_2 \wedge m_3 \\
\varphi_1 &: \ell_2 \wedge (m_{4,0,1} \vee (m_2 \wedge y_1 \leq y_2)) \\
\varphi_0 &: \ell_3
\end{aligned}$$

The resulting verification conditions are proved automatically, using the bottom-up invariants for BAKERY(2), which completes the proof.

6. Verification diagrams

Verification diagrams provide a graphical representation of the proof of a temporal property [50]. Intuitively, a verification diagram depicts the flow of the program, distinguishing the computation components that are relevant to the property of interest. Technically, we will see in Sect. 6.6 that diagrams can be thought of as ω-automata that abstract the behavior of the system.

6.1. Mutual exclusion: invariance diagrams

To prove invariants we can use *invariance diagrams*. An invariance diagram is a directed labeled graph, where each node is labeled by an assertion. Each node has an associated set of verification conditions as follows. For a φ-node (a node labeled by assertion φ) and transition τ, the verification condition associated with φ and τ is given by

$$\{\varphi\} \;\tau\; \{\varphi \vee \varphi_1 \vee \ldots \vee \varphi_k\}$$

where $\varphi_1 \ldots \varphi_k$ are the successor nodes of the φ-node. If all these verification conditions hold for a system P, we say that the diagram is *P-valid*.

A P-valid invariance diagram with nodes $\varphi_1, \ldots, \varphi_m$ proves that once we are in a state covered by the diagram, we will always stay within the set of states covered by the diagram, that is,

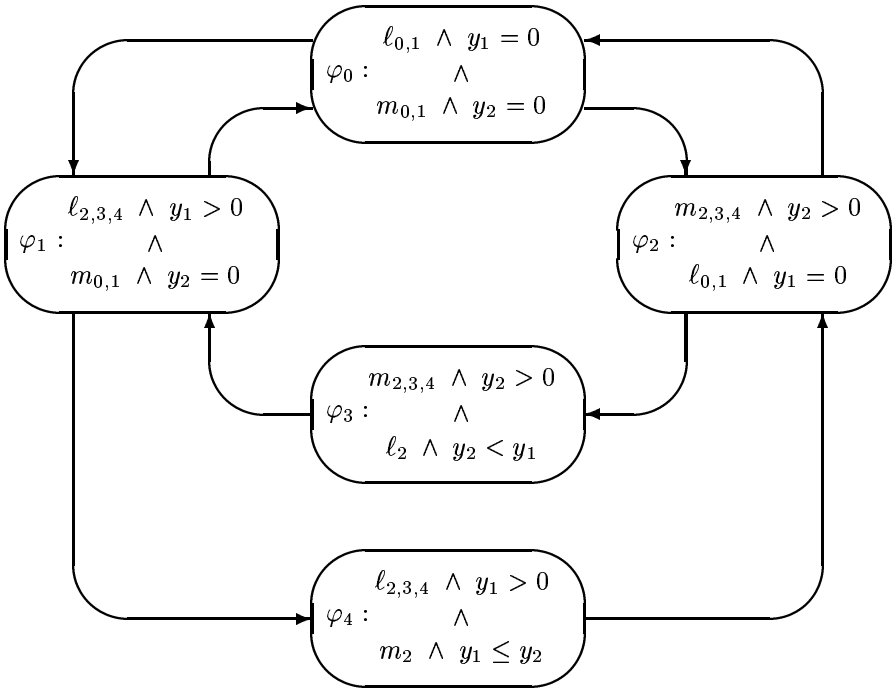

Figure 7. Invariance diagram for BAKERY(2) mutual exclusion

$$\bigvee_{j=1}^{m} \varphi_j \;\Rightarrow\; \Box \left(\bigvee_{j=1}^{m} \varphi_j \right) \quad .$$

So if we start in a state covered by the diagram, i.e.

$$\Theta \;\rightarrow\; \bigvee_{j=1}^{m} \varphi_j$$

and all states that are covered by the diagram satisfy the invariant q we want to prove, that is,

$$\bigvee_{j=1}^{m} \varphi_j \;\rightarrow\; q$$

then we can conclude that P satisfies $\Box\, q$.

Figure 7 presents a P-valid invariance diagram that proves mutual exclusion for BAKERY(2). The states corresponding to each node can be classified according to

the interest of the processes in entering the critical section (apart from the states where one process is in the critical section):

φ_0 : neither P1 nor P2 are interested

φ_1 : P1 is interested and P2 is not

φ_2 : P1 is not interested, but P2 is

φ_3 : both are interested, P2 has priority

φ_4 : both are interested, P1 has priority

6.2. Bounded overtaking: wait-for verification diagrams

Nested wait-for properties can also be verified with a special class of verification diagrams, the *wait-for diagrams*. Unlike the invariance diagrams of Sect. 6.1, which could have an arbitrary structure, wait-for diagrams must be *weakly acyclic*: node φ_i can only be connected to node φ_j if $i \geq j$. One of the nodes must be a *terminal node*, from which no edges may depart.

The verification conditions generated by wait-for diagrams are identical to those of the invariance diagrams, except for the terminal node, which does not generate any verification conditions. Similar to the invariance diagram, we say that a wait-for diagram is P-valid if all its verification conditions are P-valid.

A P-valid wait-for diagram with nodes $\varphi_n, \ldots, \varphi_0$ establishes the P-validity of the formula

$$\bigvee_{j=0}^{n} \varphi_j \Rightarrow \varphi_n \,\mathcal{W}\, \varphi_{n-1} \ldots \varphi_1 \,\mathcal{W}\, \varphi_0 \ .$$

If we can also show that

$$p \rightarrow \bigvee_{j=0}^{n} \varphi_j \qquad \text{and} \qquad \varphi_i \rightarrow q_i, \text{ for } i = 0, \ldots, n$$

are P-valid, then we can conclude the P-validity of

$$p \Rightarrow q_n \,\mathcal{W}\, q_{n-1} \ldots q_1 \,\mathcal{W}\, q_0 \ .$$

The verification conditions ensure that all the p-states are included somewhere in the diagram, and that a φ_i-interval can only lead to a φ_j-interval, where $j \leq i$. Since each φ_i strengthens the respective q_i, the system validity of the wait-for property follows.

Figure 8 shows a wait-for diagram for the one-bounded overtaking property of BAKERY(2). In this diagram we use *encapsulation conventions*: a compound node labeled by an assertion φ adds φ as a conjunct to all of its subnodes, and an edge arriving at (or departing from) a compound node represents a set of edges that arrive at (or depart from) each of its subnodes.

Note. In the special-purpose diagrams presented in this section, self-loops are implicit; thus, every node is considered its own successor.

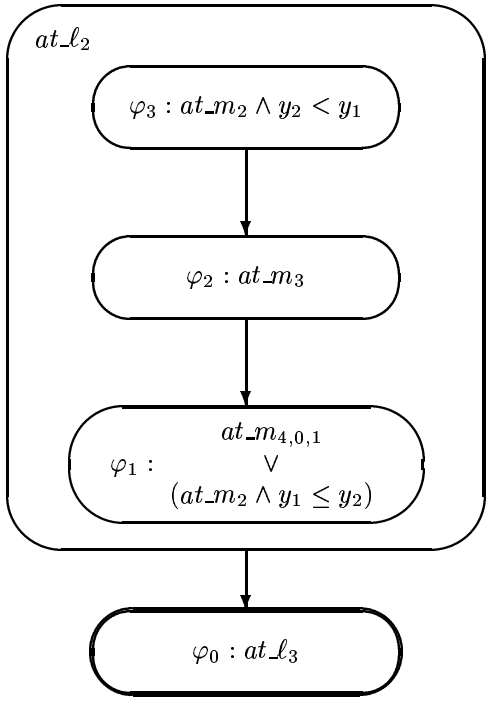

Figure 8. Wait-for diagram for Bakery[2]

6.3. Automatic generation of intermediate assertions

Wait-for formulas, and one-bounded overtaking in particular, specify safety properties, so we can use the methods from [50, 8] to automatically find the strengthened intermediate assertions needed to establish them. These methods are related to the assertion propagation used in automatic invariant generation (Sect. 4), but are specific to the property being proved. Thus, they are called *top-down* methods, as opposed to the *bottom-up* invariant generation.

A general method for generation of intermediate assertions can be extracted from the temporal formulas as illustrated in [8]. The assertions are computed from least and greatest fixpoint equations extracted from the tableau for LTL formulas, or directly from CTL formulas. An example in [8] shows how the intermediate assertions for one-bounded overtaking are found automatically. We present it here in a simplified context specific to nested wait-for formulas, generalizing the backwards propagation of Sect. 5.2.

Figure 9 presents a schematic wait-for verification diagram for proving wait-for formulas of the form

$$p \Rightarrow q_3 \,\mathcal{W}\, q_2 \,\mathcal{W}\, q_1 \,\mathcal{W}\, q_0 \ .$$

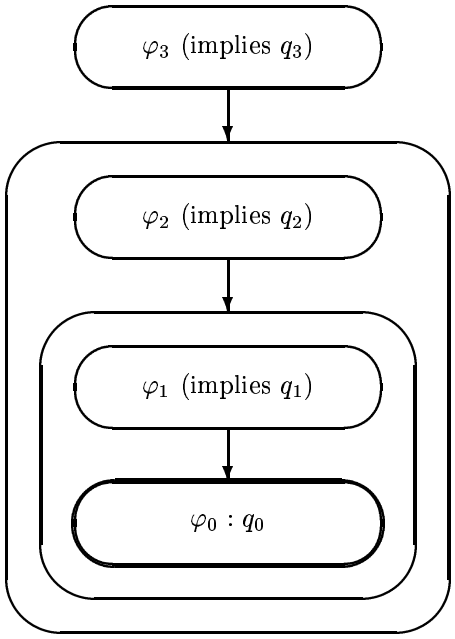

Figure 9. Schematic wait-for diagram for proving $p \Rightarrow q_3 \mathcal{W} q_2 \mathcal{W} q_1 \mathcal{W} q_0$

Strengthened assertions $\varphi_3, \varphi_2, \varphi_1, \varphi_0$ must be found such that

$$p \rightarrow (\varphi_3 \vee \varphi_2 \vee \varphi_1 \vee \varphi_0)$$

and the verification conditions

$$\{\varphi_i\}\ \tau\ \{\bigvee_{j=0}^{i} \varphi_j\} \quad \text{for every } i \in \{1,2,3\} \text{ and every } \tau \in \mathcal{T}$$

can be established (using auxiliary invariants).

We can try to find such strengthened assertions in an iterative process as follows: Let φ_i be q_i initially, and let $\tau_1 \ldots \tau_k$ be the transitions for which the verification condition $\{\varphi_i\}\ \tau\ \{\bigvee_{j=0}^{i} \varphi_j\}$ cannot be established. Then set

$$\varphi_i\ :=\ \varphi_i\ \wedge \mathit{wlpc}(\tau_1, \bigvee_{j=0}^{i} \varphi_j)\ \wedge\ \ldots\ \wedge\ \mathit{wlpc}(\tau_k, \bigvee_{j=0}^{i} \varphi_j)\ .$$

Repeat this step until all verification conditions can be established or until $p \rightarrow \bigvee_{i=0}^{n} \varphi_i$ ceases to hold, in which case the strengthening has failed.

Note that φ_0 is a terminal node, and therefore q_0 does not have to be strengthened. Furthermore, once we find a φ_1 for which $\{\varphi_1\}\,\mathcal{T}\,\{\varphi_0 \vee \varphi_1\}$ holds, both φ_1 and φ_0

can be fixed. The procedure can thus work its way backwards through the diagram, first finding a strengthened φ_1, then a φ_2, and, finally, a strengthened φ_3.

For the BAKERY(2) property

$$\textit{one-bounded}: \ell_2 \;\Rightarrow\; \neg m_3 \,\mathcal{W}\, m_3 \,\mathcal{W}\, \neg m_3 \,\mathcal{W}\, \ell_3$$

the procedure starts with the following initial values:

$$\varphi_0 : \ell_3 \quad \varphi_1 : \neg m_3 \quad \varphi_2 : m_3 \quad \varphi_3 : \neg m_3 \ .$$

We start with $i = 1$. The only verification condition that cannot be established relative to the bottom-up invariants is:

$$\{\underbrace{\neg m_3}_{\varphi_1}\}\ m_2\ \{\underbrace{\neg m_3}_{\varphi_1} \vee \underbrace{\ell_3}_{\varphi_0}\}\ .$$

We therefore set

$$\begin{array}{rcl} \varphi_1 & := & \varphi_1 \ \wedge\ \mathit{wlpc}(m_2,\ \neg m_3 \vee \ell_3) \\ & := & \neg m_3 \ \wedge\ ((m_2 \ \rightarrow\ y_1 \neq 0 \ \wedge\ y_2 \geq y_1) \ \vee\ \ell_3)\ . \end{array}$$

We check that p still implies the disjunction of $\varphi_0, \ldots \varphi_3$, and proceed to determine which verification conditions cannot be established with the new φ_1, which are

$$\{\varphi_1\}\ \ell_4\ \{\varphi_1 \vee \varphi_0\}\,, \quad \{\varphi_1\}\ \ell_1\ \{\varphi_1 \vee \varphi_0\}\,, \quad \{\varphi_1\}\ m_1\ \{\varphi_1 \vee \varphi_0\}$$

Therefore, φ_1 is strengthened into

$$\varphi_1 \ := \ \varphi_1 \wedge \mathit{wlpc}(\ell_1, \varphi_1 \vee \varphi_0) \wedge \mathit{wlpc}(\ell_4, \varphi_1 \vee \varphi_0) \wedge \mathit{wlpc}(m_1, \varphi_1 \vee \varphi_0)$$

which simplifies to

$$\neg m_3 \wedge ((m_{1,2} \rightarrow y_1 \neq 0) \wedge (m_2 \rightarrow y_2 \geq y_1) \wedge \neg \ell_{1,4} \ell_3)\ .$$

For example, $\mathit{wlpc}(\ell_1, \varphi_1 \vee \varphi_0) = \neg \ell_1$. Now the verification conditions

$$\{\varphi_1\}\ \ell_3\ \{\varphi_1 \vee \varphi_0\}\,, \quad \{\varphi_1\}\ \ell_0\ \{\varphi_1 \vee \varphi_0\}\,, \quad \{\varphi_1\}\ m_0\ \{\varphi_1 \vee \varphi_0\}$$

do not hold, so we strengthen φ_1 into

$$\varphi_1 \ := \ \varphi_1 \wedge \mathit{wlpc}(\ell_3, \varphi_1 \vee \varphi_0) \wedge \mathit{wlpc}(\ell_0, \varphi_1 \vee \varphi_0) \wedge \mathit{wlpc}(m_0, \varphi_1 \vee \varphi_0)$$

which simplifies to

$$\neg m_3 \wedge (m_{0,\ldots,2} \rightarrow y_1 \neq 0) \wedge (m_2 \rightarrow y_2 \geq y_1) \wedge \neg \ell_{0,1,3,4}\ .$$

It is still necessary to strengthen φ_1 since now

$$\{\varphi_1\}\ m_4\ \{\varphi_1 \vee \varphi_0\}$$

is not established. After this strengthening we get the final formula φ_1 below. The intermediary assertions φ_2 and φ_3 are strengthened in a similar way. The procedure terminates successfully with the result:

$$\begin{array}{l}\varphi_0 : \ \ell_3 \\ \varphi_1 : \ \ell_2 \ \wedge \ \neg m_3 \ \wedge \ (m_2 \rightarrow y_2 \geq y_1) \ \wedge \ y_1 \neq 0 \\ \varphi_2 : \ \ell_2 \ \wedge \ m_3 \\ \varphi_3 : \ \ell_2 \ \wedge \ \neg m_3\end{array}$$

Since all the verification conditions associated with the wait-for diagram of Fig. 9 are now P-valid, we have a proof of one-bounded overtaking.

6.4. Accessibility: chain diagrams

One-bounded overtaking guarantees that process P2 will enter the critical section at most once before P1 is allowed to do so. However, it does not guarantee that P1 will eventually enter the critical section if it is interested: a wait-for formula allows intermediate intervals to have zero duration or extend to infinity. The *accessibility* property states that P1 will eventually reach its critical section, once it is at ℓ_1:

$$acc : \ \ell_1 \Rightarrow \Diamond \ell_3$$

or, in STeP format,

```
l1 ==> <>l3
```

The properties we have considered so far, mutual exclusion and one-bounded overtaking, are safety properties. Accessibility is a *response* property. As mentioned in Sect. 2.2, the proof of response properties relies on the fairness requirements for computations. Response properties state that eventually something "good" will happen; but by repeatedly taking the idling transition, a run of the system could forever stay in the same state, if fairness is not enforced.

Similar to the verification rules for invariance and wait-for properties, verification rules also exist for response properties [45]. However, we have found that verification diagrams are much more convenient to prove response properties.

We use a *chain diagram* to prove accessibility for BAKERY(2). These diagrams are similar to wait-for diagrams, except for their ability to represent fairness. Each edge in a chain diagram may be labeled by a fair (just or compassionate) transition.

The diagram must be *weakly acyclic* in the unlabeled edges: if an unlabeled edge connects node φ_i to node φ_j then $i \geq j$. It should also be *strictly acyclic* in the labeled edges: if a labeled edge connects node φ_i to node φ_j then $i > j$. In addition, we require that every node φ_i should have a labeled edge departing from it, except for φ_0, which is a terminal node and has no outgoing edges. A transition that labels an edge departing from φ_i is called a *helpful transition* for φ_i.

A chain diagram has the same verification conditions as a wait-for diagram. Additional verification conditions are associated with the labels on the edges: For each

φ-node and transition τ such that τ labels some outgoing edge from φ, we associate the verification conditions

$$\varphi \rightarrow \mathit{enabled}(\tau)$$

and

$$\{\varphi\}\,\tau\,\{\varphi_1 \vee \ldots \vee \varphi_k\}$$

where $\varphi_1 \ldots \varphi_k$ are the successor nodes of φ connected by an edge labeled by τ. The verification conditions assert that τ is guaranteed to be enabled on φ, and can be taken along one of the τ-labeled edges.

A P-valid chain diagram establishes that the response formula

$$\bigvee_{j=0}^{m} \varphi_j \Rightarrow \Diamond\, \varphi_0$$

is P-valid. To establish

$$p \Rightarrow \Diamond\, q$$

we have to prove in addition

$$p \rightarrow \bigvee_{j=0}^{m} \varphi_j \quad \text{and} \quad \varphi_0 \rightarrow q \ .$$

The chain diagram in Fig. 10 represents a proof of accessibility for BAKERY(2). It is a graphical representation of the intermediate assertions that a computation must go through, and the helpful transitions on which it depends to reach the goal assertion φ_0.

All verification conditions are proven automatically by STeP, using the bottom-up invariants.

6.5. Accessibility: rank diagrams

Program BAKERY_VIS(2), shown in Fig. 11, is a variation on the original BAKERY(2), where process P1 needs to complete n visits, and does not have to visit the critical section afterwards. Thus, the accessibility property we want to prove for this program is

$$\mathit{visits}: \ \Diamond(n = 0) \ .$$

The chain diagram used to prove accessibility for BAKERY(2) can be used here to prove successful completion of one visit. Intuitively, we would need to connect n copies of this diagram to establish successful completion of n visits. However, this is not practical, since we do not know *a priori* the value of n, and it would be tedious to produce a diagram for each value of n.

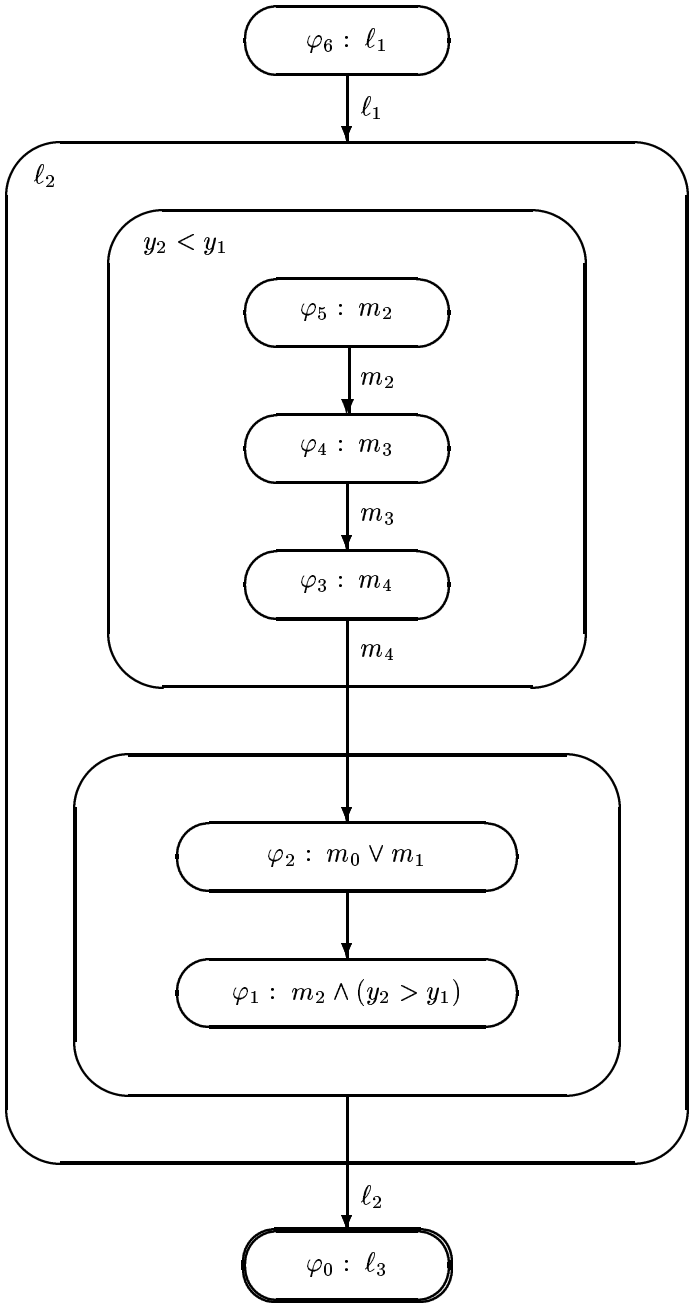

Figure 10. Chain diagram for proving accessibility for BAKERY(2)

$$\begin{array}{lll} \textbf{in} & n & : \textbf{integer where } n > 0 \\ \textbf{local} & y_1, y_2 & : \textbf{integer where } y_1 = 0 \ \wedge \ y_2 = 0 \end{array}$$

$$\left[\begin{array}{l} \ell_0\text{: } \textbf{while } n > 0 \textbf{ do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } y_1 := y_2 + 1 \\ \ell_2\text{: } \textbf{await } (y_2 = 0 \vee y_1 \leq y_2) \\ \ell_3\text{: } \textbf{critical} \\ \ell_4\text{: } y_1 := 0 \\ \ell_5\text{: } n := n - 1 \end{array}\right] \\ \ell_6\text{:} \end{array}\right] \quad || \quad \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_0\text{: } \textbf{noncritical} \\ m_1\text{: } y_2 := y_1 + 1 \\ m_2\text{: } \textbf{await } (y_1 = 0 \vee y_2 < y_1) \\ m_3\text{: } \textbf{critical} \\ m_4\text{: } y_2 := 0 \end{array}\right] \end{array}\right]$$

–P1– –P2–

Figure 11. Program BAKERY_VIS(2) with n visits

A solution to this problem is to use well-founded induction within diagrams. This is done by labeling nodes with ranking functions over a well-founded domain $\mathcal{D}$ (see Sect. 2) and removing the requirement that the diagram be acyclic. The ranking functions that label the nodes generate the following verification conditions for the diagram: Consider a nonterminal node labeled by assertion φ and ranking function δ. Let $\varphi_1, \ldots, \varphi_k$ be the successor nodes of φ, and $\delta_1, \ldots, \delta_k$ be their respective ranking functions. Then we associate with φ the following verification condition for every transition τ:

$$\{\varphi \wedge \delta = u\} \ \tau \ \{(\varphi \wedge u \succeq \delta) \vee (\varphi_1 \wedge u \succ \delta_1) \vee \ldots \vee (\varphi_k \wedge u \succ \delta_k)\}.$$

Figure 12 shows a chain diagram that proves the property $\diamondsuit(n = 0)$. The ranking function associated with each node φ_i is the pair (n, i), lexicographically ordered. Each traversal of an edge decreases i and keeps n constant, except for the edge from φ_1 to φ_9. This edge increases i, but decreases n, thereby also decreasing the ranking function.

6.6. Generalized verification diagrams

Program BAKERY_LAZY(2) in Fig. 13 is a variation on BAKERY_VIS(2) of the previous section, where the **critical** statement at m_3 has been replaced by an **idle** statement. The significance of this change is that the transition associated with the **idle** statement has no fairness (that is, it behaves like the **noncritical** statement), so we cannot use m_3 as a helpful transition to prove completion of visits. In fact, completion of visits, as defined before, is not valid for this program, since there is no assurance that process P2 will always leave location m_3: it may stay there indefinitely, prohibiting P1 from entering the critical section.

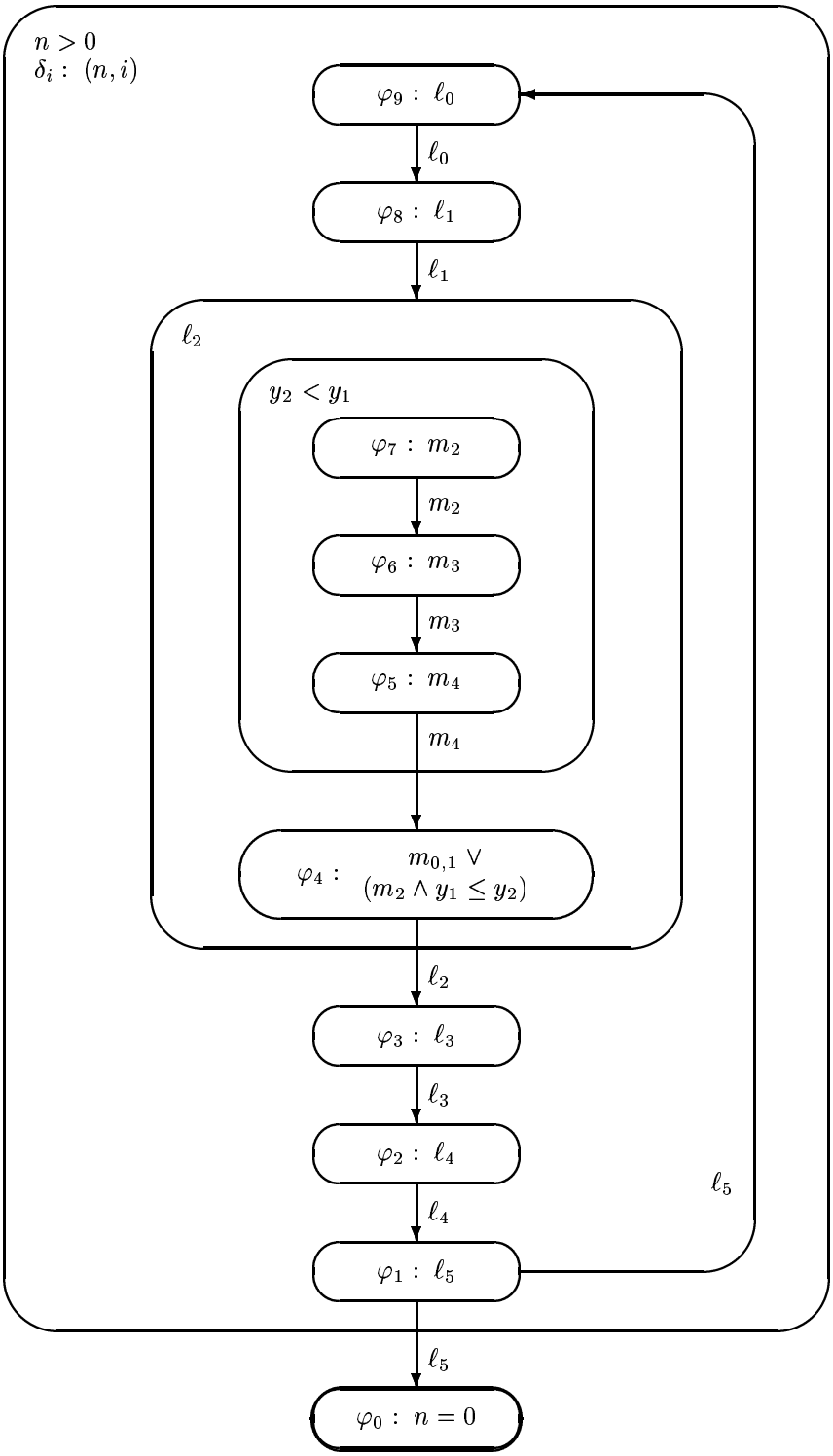

Figure 12. Chain diagram with ranking functions for proving completion of visits

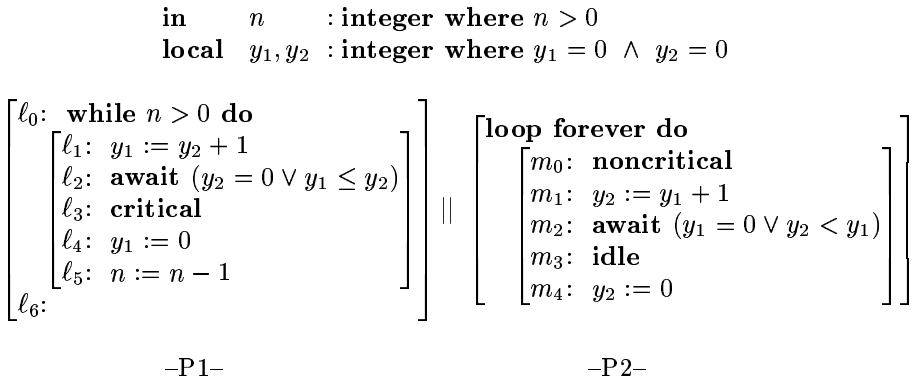

Figure 13. Program BAKERY_LAZY(2) with a lazy customer

The best we can prove for this program is a conditional completion of visits, expressed by the formula

$$\mathit{cond\_visits} : (\Box \Diamond \neg m_3) \ \rightarrow \ \Diamond\, n = 0$$

stating that if P2 is always eventually out of the critical section (m_3) then P1 can complete its visits. While it is possible to develop specialized rules and diagrams for this class of properties, as was the case with the other properties we have considered so far, it is more efficient to develop a general notion of diagram that can be applied to arbitrary properties. This is achieved by *generalized verification diagrams* (GVD's) [16], which are applicable to any state-quantified temporal logic formula.

Like temporal formulas and ω-automata (see [61]), GVD's have an associated set of computations, constrained by an *acceptance condition*. Verification conditions similar to those for invariance diagrams establish that all runs of the system $\mathcal{S}$ are represented in the diagram. Fairness properties and ranking functions justify the GVD acceptance condition. Together, these verification conditions prove that all computations of $\mathcal{S}$ are accepted by the GVD.

To show that all computations accepted by the GVD satisfy the temporal property φ, hence showing that $\mathcal{S}$ satisfies φ, a propositional node labeling relates the nodes of the diagram with the literals of φ. Given this labeling, checking that the GVD computations satisfy φ can be done algorithmically—it is a decidable language containment check between two ω-automata.

Definition: We now define GVD's more formally, following the presentation of [43, 60]. A GVD is a directed labeled graph where each node n_i is labeled by an assertion φ_i, and edges are now labeled by sets of transitions. A set of nodes $\mathcal{I}$ is marked as *initial*.

A GVD has a Müller acceptance condition $\mathcal{F}$, which is a set of sets of nodes. (The original GVD's of [16] have *edge-Streett* acceptance conditions, which are more succinct but less intuitive.) For a path π through the diagram, let $inf(\pi)$ be the set of nodes that appear infinitely often in π. Such a set will be a *strongly connected subgraph* (SCS) in the GVD. A path π satisfies the acceptance condition $\mathcal{F}$ iff $inf(\pi) \in \mathcal{F}$.

The assertion labeling a node n is indicated by $\mu(n)$. For a node n, the set of successor nodes of n is $succ(n)$. For a set $S : \{n_1, \ldots, n_k\}$ of nodes, $\mu(S)$ stands for $\mu(n_1) \vee \ldots \vee \mu(n_k)$.

A GVD Ψ has the following associated verification conditions, relative to a fair transition system:

- **Initiality:** At least one initial node satisfies the initial condition:

$$\Theta \rightarrow \bigvee_{n_i \in \mathcal{I}} \mu(n_i) \ .$$

- **Consecution:** Every successor of a $\mu(n_i)$-state is a $\mu(n_j)$-state, where n_j is a successor node of n_i. For every transition τ, and every node n_i,

$$\{\mu(n_i)\} \ \tau \ \{\mu(n_1) \vee \ldots \vee \mu(n_k)\}$$

  where $n_1, \ldots, n_k$ are all the successor nodes of n_i.

- **Acceptance:** We must show that every computation of the system has an accepting path through the diagram. We do this by showing that each SCS S that is not accepting (that is, not in $\mathcal{F}$) has one of the following properties:

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

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

    This means that τ is enabled and can leave the SCS at all nodes.

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

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

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

$$\mu(n) \ \rightarrow \ enabled(\tau) \ .$$

    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.

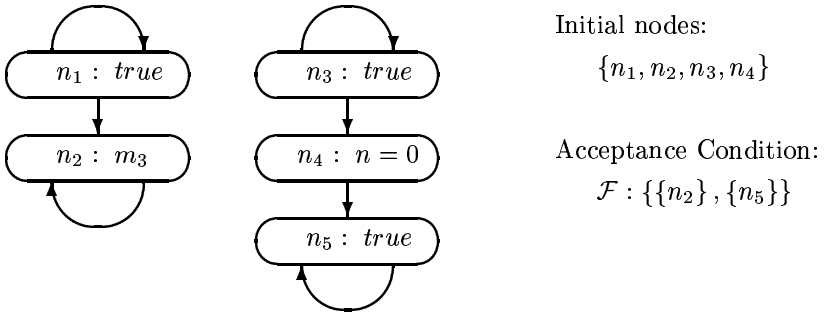

Figure 14. Müller automaton for $\Diamond \Box m_3 \vee \Diamond (n = 0)$

– $S : \{n_1, \ldots, n_k\}$ 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* E of edges in S (that is, every loop in S contains some edge in E) such that for each edge $(n_1, n_2) \in E$ and every transition τ,

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

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

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

This means that there is no computation that stays within S from some point onwards: it would have to traverse at least one of the edges in E infinitely often, which contradicts the well-foundedness of the ranking functions.

The above verification conditions ensure that the computations of the system are a subset of the computations of Ψ. If in addition we can show that the computations of Ψ are a subset of those that satisfy the property φ, we can conclude that the system satisfies φ.

For this, we add a *propositional label* p_i to each node φ_i, and add one last set of verification conditions:

- **Propositions:** $\mu(n_i) \rightarrow p_i$ for each node n_i.

The language inclusion between Ψ (using the propositional labeling only) and φ can then be checked algorithmically (see Sect. 3.6).

Example: Figure 14 shows a Müller automaton for formula *cond_visits*. The acceptance list requires that any sequence of states that satisfies the property has to end up in n_2 or n_5. Figure 15 presents the verification diagram that represents a proof of the property. The only accepting SCSs of this diagram

are n_6 and n_0. Initiality and Consecution are easily proven for this diagram. To show Acceptance we must show that every non-accepting SCS either has a fair exit or is well-founded. Clearly all non-accepting single-node SCSs have a fair exit transition, which is shown in the diagram. The SCS $\{n_1 \dots n_9\}$ can be shown to be well-founded with ranking function n labeling all nodes, and cut-set $\langle n_1, n_9 \rangle$. None of the transitions increase n and the transition that can be taken along $\langle n_1, n_9 \rangle$, that is ℓ_5, strictly decreases n.

Finally, the propositional labeling enables us to check that the language of the diagram is included in the property.

A valid generalized verification diagram can itself be seen as a weakly-preserving abstraction of the system (c.f. Sect. 7) [43, 62].

7. Abstracting and model checking BAKERY(2)

Abstraction allows proving properties over an abstract system and then infering the validity of a related property over a larger or more complex concrete system. This is most attractive when the abstract system is finite-state while the concrete system is infinite-state: abstraction enables the use of automatic methods that are not applicable to the concrete system.

BAKERY(2) is infinite-state, so the finite-state model-checking techniques described in Sect. 3.6 cannot be applied. However, it is straightforward to construct a finite-state abstraction, BAKERY-ABSTRACT, shown in Fig. 16, that preserves the properties of interest. In BAKERY-ABSTRACT, statements ℓ_1, ℓ_2, ℓ_4, m_1, m_2, and m_4 have been changed, but the structure is otherwise that of program BAKERY(2) of Fig. 1.

The computations of BAKERY-ABSTRACT and BAKERY(2) are closely related. The variable b_1 has been introduced in place of testing whether $y_1 = 0$; likewise, b_2 corresponds to the test $y_2 = 0$, and b_3 to whether $y_1 \leq y_2$. That is, b_1 is true in a computation of BAKERY-ABSTRACT if and only if $y_1 = 0$ in the corresponding computation of BAKERY(2), and similarly for b_2 and b_3.

Using the method of [21], STeP automatically generates this abstract program given the user-provided *assertion basis*

$$\{b_1 : y_1 = 0, b_2 : y_2 = 0, b_3 : y_1 \leq y_2\},$$

indicating that the infinite-domain variables y_1 and y_2 are removed, while the finite-state control variables are retained.

The abstraction procedure guarantees *weak preservation* of LTL properties: if a property holds for the abstract system (BAKERY-ABSTRACT), then the corresponding property will hold for the original one (BAKERY(2)). This is justified by the fact that BAKERY(2) is a *fair refinement* of BAKERY-ABSTRACT via the refinement relation

$$\gamma : \ (b_1 \leftrightarrow y_1 = 0) \ \wedge \ (b_2 \leftrightarrow y_2 = 0) \ \wedge \ (b_3 \leftrightarrow y_1 \leq y_2) \ .$$

For every computation $\sigma^{\mathcal{C}}$ of BAKERY(2) there is a computation $\sigma^{\mathcal{A}}$ of BAKERY-ABSTRACT, such that $\gamma\left(\sigma_i^{\mathcal{C}}, \sigma_i^{\mathcal{A}}\right)$ holds for each position $i \geq 0$. It is also said that

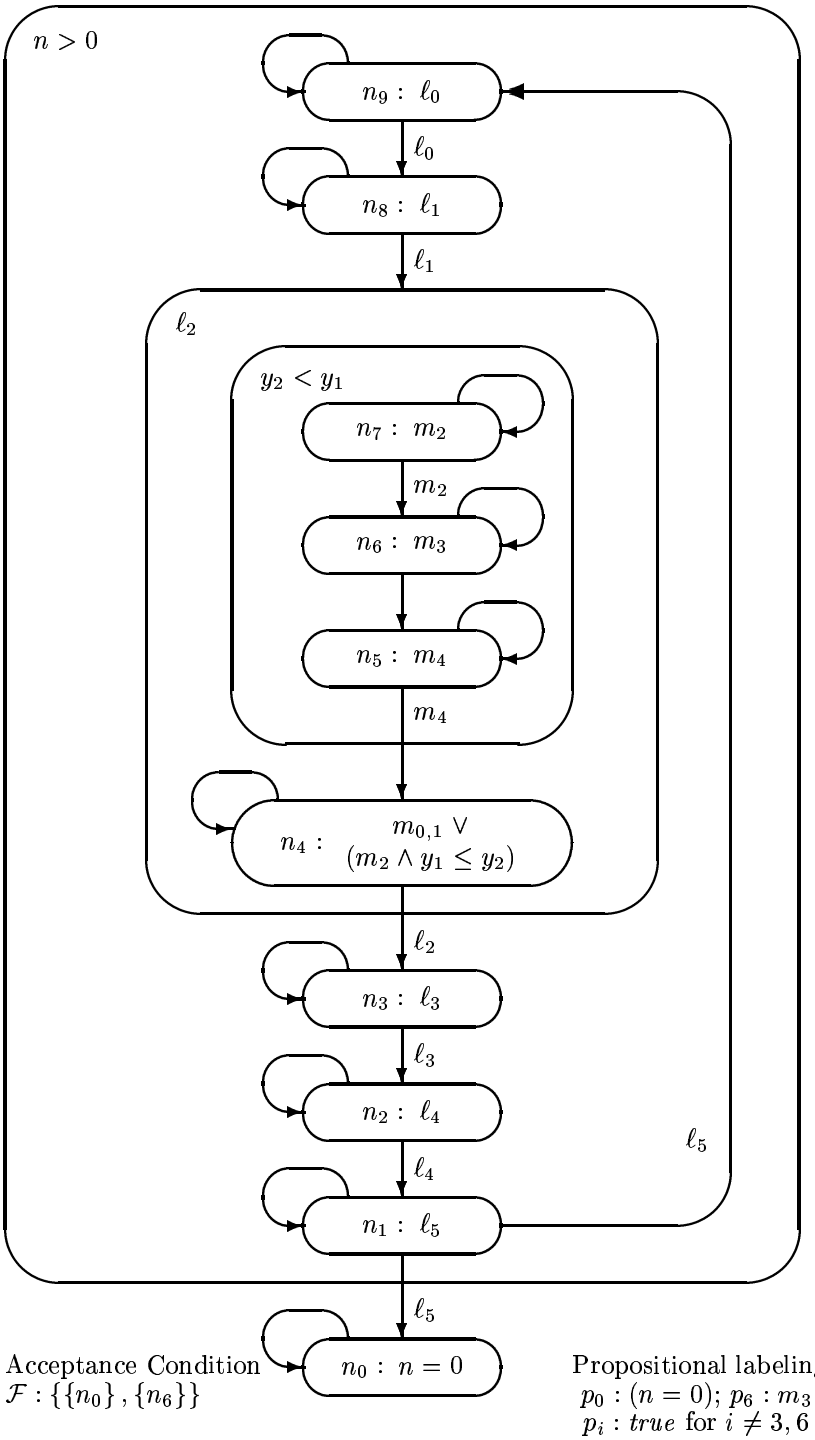

Figure 15. GVD proving conditional completion of visits: $\Box \Diamond \neg m_3 \rightarrow \Diamond (n = 0)$

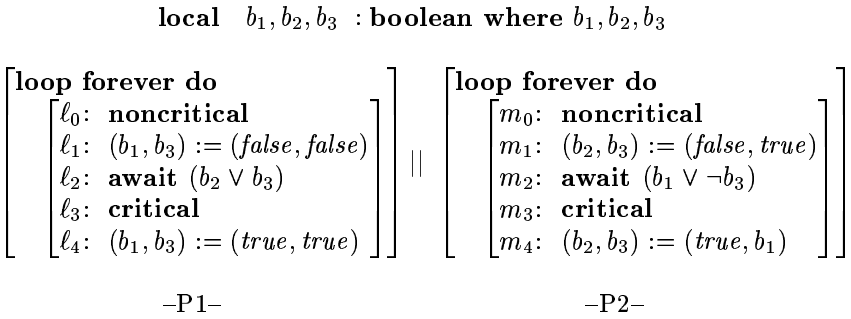

Figure 16. Program BAKERY-ABSTRACT

BAKERY-ABSTRACT *fairly* γ*-simulates* BAKERY(2). Thus, if a temporal property is true for all computations of BAKERY-ABSTRACT, the corresponding one will hold for all computations of BAKERY(2).

Since BAKERY-ABSTRACT is finite-state, we can use STeP's model checker to prove the main requirements: mutual exclusion, bounded overtaking, and accessibility. Since the model checker accepts temporal specifications in any format, we can give it the conjunction of all requirements:

$$\psi \; : \; \begin{array}{ll} & \Box\, \neg(\ell_3 \wedge m_3) \\ \wedge & (\ell_2 \;\Rightarrow\; \neg m_3 \,\mathcal{W}\, m_3 \,\mathcal{W}\, \neg m_3 \,\mathcal{W}\, \ell_3) \\ \wedge & (\ell_1 \;\Rightarrow\; \Diamond\, \ell_3) \end{array} \tag{3}$$

which is model checked by STeP in a fraction of a second. This implies the validity of mutual exclusion, one-bounded overtaking and accessibility for the infinite-state concrete system, BAKERY(2).

For more on simulation and refinement, see e.g. [33, 20, 41]. Other approaches to the generation of abstract finite-state systems are presented in [29, 3]. As with the invariant generation methods of Sect. 4, the underlying theory is based on abstract interpretation [22]—see, for instance, [24, 8, 62].

8. Atomic BAKERY(N)

In many applications, an unknown or large number of processes compete for access to a critical section, rather than only two. We would like to specify, and verify, such protocols for an arbitrary number of processes.

An N-process generalization of BAKERY(2) is shown in Figure 17, following the presentation of [57]. This is a *parameterized program*, where N copies of a process are composed, for an arbitrary N. Each process is indexed by a variable i, which ranges between 1 and N; local variables for each process are expressed as arrays, e.g. $y[i]$ is the value of variable y for process i. Similarly, each control location variable will now be an array, where e.g. $\ell_k[i]$ is *true* iff process i is at location ℓ_k.

$$
\begin{array}{ll}
\textbf{in} & N : \textbf{integer where } N > 0 \\
\textbf{local} & y : \textbf{array}\,[1..N]\ \textbf{of integer where } \forall i : [1..N]\,.\, y[i] = 0 \\
\textbf{value} & \max : \textbf{array}\,[1..N]\ \textbf{of } (\textbf{integer} \rightarrow \textbf{integer})
\end{array}
$$

$$
\begin{array}{c} N \\ \| \\ i = 1 \end{array} :: \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} \ell_0\text{:} & \textbf{noncritical} \\ \ell_1\text{:} & y[i] := 1 + \max(y) \\ \ell_2\text{:} & \textbf{await } \forall j : [1..N].(i \neq j \ \rightarrow\ y[j] = 0 \ \vee\ y[i] \leq y[j]) \\ \ell_3\text{:} & \textbf{critical} \\ \ell_4\text{:} & y[i] := 0 \end{array} \right] \end{array} \right]
$$

Figure 17. Atomic BAKERY(N)

STeP parses parameterized programs such as BAKERY(N) into parameterized fair transition systems, where each transition is now indexed by i, $1 \leq i \leq N$ [6].

In this program, max is declared as an uninterpreted function symbol. In the specification we axiomatize the relevant properties of max:

$$
\begin{array}{ll}
\text{Axiom 1} & : \ \Box\, \forall i : [1..N]\ .\ \max(y) \geq y[i] \\
\text{Axiom 2} & : \ \Box\, \exists i : [1..N]\ .\ \max(y) = y[i]
\end{array}
$$

Deductive verification can be used to verify such parameterized systems [49], whereas finite-state methods can do so only for specific values of N. It is, however, a good idea to automatically model check the system for such particular values, if possible, to quickly find obvious bugs. For this, STeP allows the user to instantiate N and model check the resulting system.

Unfortunately, this approach still has some limitations, evidenced in our example. After instantiating N the system may still be infinite state: instantiating N by 2 gives us BAKERY(2), which we saw had infinitely many reachable states. Also, using underspecified functions such as max prevents the model checker from evaluating the effect of transitions (in our case, transition ℓ_1).

Parameterization introduces quantifiers in verification conditions: to establish $\{\psi\}\,\tau[i]\,\{\varphi\}$ for each i, we universally quantify over i, $1 \leq i \leq N$.

The parameterized program BAKERY(N) is a simplification of the finer-grained versions introduced by Lamport in [37, 38]. The Bakery algorithm has been used as an example in many other verification frameworks, including, for instance, the refinement of abstract state machines in [13]. We have also used STeP for a machine-checked analysis of these more complicated versions, and use *diagram induction* in [52]. For conciseness and clarity, we will only describe the verification of the simpler version, which we call the *atomic* one.

8.1. Mutual exclusion

Mutual exclusion for BAKERY(N) can be expressed as:

$$mux \ : \ \Box \forall i, j : [1..N] \ . \ \ell_3[i] \ \wedge \ \ell_3[j] \rightarrow i = j \ .$$

Note. Quantifiers in STeP have maximal scope.

We need auxiliary invariants to prove this property, as was the case for BAKERY(2). However, STeP's invariant generation methods for parameterized systems are more limited. The following invariants were found in an interactive fashion: in the proof of desired invariants, the need for more auxiliary invariants was identified.

$$\begin{aligned}
\varphi_A \ &: \ \Box \forall i : [1..N] \ . \ y[i] \geq 0 \\
\varphi_B \ &: \ \Box \forall i : [1..N] \ . \ \ell_{2..4}[i] \rightarrow y[i] > 0 \\
\varphi_C \ &: \ \Box \forall i, j : [1..N] \ . \ i \neq j \ \wedge \ \ell_{2..4}[i] \ \wedge \ \ell_{2..4}[j] \ \rightarrow \ y[i] \neq y[j] \\
\varphi_D \ &: \ \Box \forall i, j : [1..N] \ . \ \ell_2[i] \wedge \ell_{3,4}[j] \ \rightarrow \ y[j] < y[i]
\end{aligned}$$

Each invariant is proved using `B-INV`, with the preceding invariants as background properties. The proof of some of the verification conditions require the use of the interactive prover to instantiate quantified variables. In the proof of properties φ_C and φ_D the verification condition for ℓ_1 requires the addition of the axioms for max and their instantiation by the (skolemized) i and j. The verification condition for ℓ_2 in the proof of φ_D requires the addition of φ_B and φ_C, with simple instantiations. Using these invariants, *mux* can be proved using `B-INV` with ℓ_2 being the only verification condition that requires interaction.

8.2. One-bounded overtaking

One-bounded overtaking for BAKERY(N) can be expressed by

$$\begin{aligned}
overtaking \ : \ &\Box \forall i, j : [1..N] \ . \\
&(\ell_2[i] \rightarrow \ \neg \ell_3[j] \, \mathcal{W} \, \ell_3[j] \, \mathcal{W} \, \neg \ell_3[j] \, \mathcal{W} \, \ell_3[i])
\end{aligned}$$

Similarly to BAKERY(2), property *overtaking* states that if a process i is waiting to enter the critical section, any other process j can only enter the critical section once before i does.

To prove property *overtaking* we need two additional invariants, which are generated automatically by STeP as bottom-up local invariants:

$$\begin{aligned}
\varphi_E \ &: \ \Box \forall i : [1..N] \ . \ \ell_{0,1}[i] \ \rightarrow \ y[i] = 0 \\
\varphi_F \ &: \ \Box \forall i : [1..N] \ . \ \ell_{0..4}[i]
\end{aligned}$$

The proof of property *overtaking* uses the wait-for diagram shown in Figure 18, where i and j have been skolemized into I and J. Again, the verification conditions associated with the diagram are universally quantified over the process index i, for $1 \leq i \leq N$.

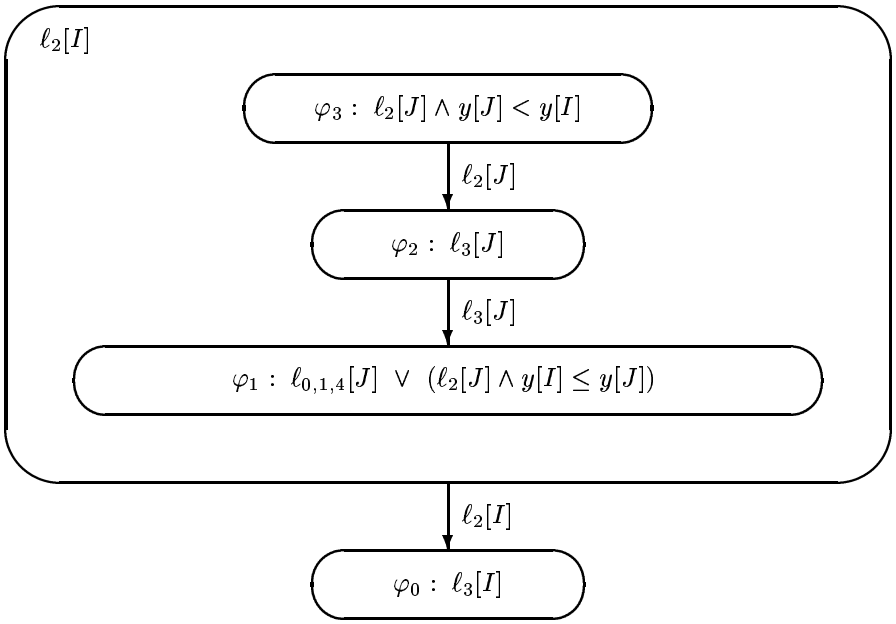

Figure 18. Wait-for diagram for BAKERY(N)

9. Conclusions

STeP presents a collection of tools for verifying linear-time temporal properties of fair transition systems, including model checking, verification rules, verification diagrams, and abstraction.

9.1. Limitations

A wide range of systems can in principle be modeled as fair transition systems. However, the convenience with which a given system can be verified with STeP is determined by how naturally the system can be described as a set of transitions. In particular, it is often difficult to conveniently represent and verify large synchronous systems in this way. On the other hand, relatively small but intricate concurrent protocols appear to be the most amenable to analysis with STeP.

Programming languages such as SPL facilitate the description of transition systems. However, we note that SPL does not include function or procedure calls, although STeP has macro functionalities that can mimick procedure calls when they are not recursive.

The assertion language of STeP is based on first-order logic, and does not offer the same expressiveness and flexibility of higher-order logic theorem-provers; these,

on the other hand, require more complex deductive support. Proofs in STeP can be saved and re-run as tactics, but the proof management is not as flexible or robust as that of PVS.

STeP includes basic types in its assertion language, and system variables are similarly typed. However, the value of ill-typed expressions is left underspecified, as in most first-order theorem provers. Run-time type errors must be checked independently. STeP generates type check obligations which, if proved, ensure the absence of run-time errors in the system being verified. (See [40] for a discussion of types in formal specifications.) Some datatypes are only partially supported by the theorem proving and model checking components.

STeP is mostly intended as a tool for teaching and research, and thus does not provide some facilities that an "industrial-strength" system should. For instance, checking the consistency of axiomatizations is left to the user. Furthermore, the syntax for transitions is general enough that users may inadvertently specify inconsistent transitions (i.e. equivalent to *false*). For real-time and hybrid systems, *non-Zenoness* of system specifications should be checked to ensure that they are implementable [10].

As mentioned in Sect. 2, flexible temporal quantification is not addressed by the verification rules, which cover state-quantified temporal formulas only. Branching-time temporal properties are not supported in the specification language. However, the system can be used to verify branching-time properties provided that the appropriate verification conditions are constructed manually (e.g. [10]).

9.2. Related work

Verification Systems: Deductive verification can also be conducted in powerful general-purpose theorem-proving environments such as PVS [55] and HOL [28]. Most of these systems now include model checking procedures to evaluate μ-calculus formulas and finite-state verification conditions. These tools require encoding the system semantics and property specification language within the logic of the theorem prover, giving extra flexibility at the cost of a layer of encoding.

Tableau-based (deductive) proof systems for the verification of general properties of infinite-state systems are presented in [15, 26], including branching-time properties. The Temporal Logic of Actions (TLA) [39] provides a verification framework based on linear-time temporal logic, where systems and properties are expressed as formulas in a single general logical formalism. Verification then corresponds to showing implication between formulas.

Theorem Proving: The decision procedures used in STeP have their roots in the work of [54, 59]. Similar procedures can be found in verification systems and theorem provers such as PVS and NQTHM [14].

The validity checking procedure used in STeP is described in [11] and [5]. Similar procedures can be found in the Stanford Validity Checker (SVC) [2], which is restricted to the ground case (i.e. does not include first-order quantification), and the Extended Static Checking (ESC) tool [25], which does handle quantifiers.

Model Checking: Efficient finite-state model checkers include SMV [53], which performs CTL symbolic model checking, SPIN [31] (on-the-fly model checking for LTL), and COSPAN [30] (automata-theoretic).

9.3. Applications

STeP has been used successfully in introductory graduate-level courses on formal methods and temporal verification of reactive systems. The STeP documentation includes exercises and tutorials based on the textbook [50].

Properties of a parameterized fault-tolerant leader-election algorithm are verified in [9], while modularity is explored in [44]. The use of STeP in the modular verification of real-time systems is described in [10]. The verification of real-time and hybrid systems using STeP is further discussed in [51, 60]. Accessibility for the molecular version of BAKERY(N) is verified in [52], using induction on verification diagrams.

9.4. Obtaining STeP

Being intended primarily as a research tool, the STeP implementation is constantly updated and upgraded to reflect new directions in research. We are about to release STeP 2.0, which features all the verification approaches presented in this paper, with the exception of the intermediate assertion generation of Sect. 6.3.

The latest release of the system, the documentation, and a detailed tutorial, are freely available on the web. For more information, see the STeP home page,

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

Acknowledgements

Help in the design and implementation of STeP has been provided by Eddie Chang and Mark Pichora. Andrew Miller helped implement the STeP 2.0 interface. Mark Stickel provided AC-unification utilities and feedback.

We thank Francesc Babot, Ralph Jeffords, Arjun Kapur, Uri Lerner, Raya Leviathan, Ka Fai Lo, Jonathan Ostroff, Amir Pnueli, Calogero Zarba, and the students of CS256 at Stanford for their feedback on STeP.

We also thank the anonymous referees for their comments, suggestions and corrections concerning this paper.

References

1. R. Alur and T. A. Henzinger, editors. *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*. Springer-Verlag, July 1996.
2. C. Barrett, D. L. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *1st Intl. Conf. on Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, Nov. 1996.

3. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [32], pages 319–331.
4. S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [1], pages 323–335.
5. N. S. Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Computer Science Department, Stanford University, Nov. 1998.
6. N. S. Bjørner, A. Browne, E. S. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, Nov. 1995.
7. 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 Alur and Henzinger [1], pages 415–418.
8. N. S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, Feb. 1997. Preliminary version appeared in 1st *Intl. Conf. on Principles and Practice of Constraint Programming*, vol. 976 of LNCS, pp. 589–623, Springer-Verlag, 1995.
9. N. S. Bjørner, U. Lerner, and Z. Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Intl. Conf. on Temporal Logic*. Kluwer, 1997. To appear.
10. N. S. Bjørner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using STeP. Technical Report STAN-CS-TR-98-1616, Stanford University, Jan. 1998. To appear in *Theoretical Computer Science*. Preliminary version appeared in *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pp. 22–43. Springer-Verlag, May 1997.
11. N. S. Bjørner, M. E. Stickel, and T. E. Uribe. A practical integration of first-order reasoning and decision procedures. In *Proc. of the* 14^{th} *Intl. Conference on Automated Deduction*, volume 1249 of *LNCS*, pages 101–115. Springer-Verlag, July 1997.
12. E. Börger, editor. *Specification and Validation Methods*. International Schools for Computer Scientists. Oxford University Press, 1994.
13. E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery algorithm: yet another specification and verification. In Börger [12], pages 231–243.
14. R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.
15. J. C. Bradfield and C. Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96(1):157–174, Apr. 1992.
16. 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*, volume 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.
17. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
18. E. S. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In W. Kuich, editor, *Proc. 19th Intl. Colloq. Aut. Lang. Prog.*, volume 623 of *LNCS*, pages 474–486. Springer-Verlag, 1992.
19. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
20. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5):1512–1542, Sept. 1994.
21. M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Hu and Vardi [32], pages 293–304.
22. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 238–252. ACM Press, 1977.
23. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among the variables of a program. In 5^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 84–97, Jan. 1978.
24. D. R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, July 1996.

25. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq SRC, Dec. 1998.
26. L. Fix and O. Grumberg. Verification of temporal properties. *J. Logic and Computation*, 6(3):343–362, 1996.
27. S. M. German and B. Wegbreit. A Synthesizer of Inductive Assertions. *IEEE transactions on Software Engineering*, 1(1):68–75, Mar. 1975.
28. M. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
29. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9^{th} Intl. Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.
30. R. Hardin, Z. Har'El, and R. Kurshan. COSPAN. In Alur and Henzinger [1], pages 423–427.
31. G. J. Holzmann and D. Peled. The state of SPIN. In Alur and Henzinger [1], pages 385–389.
32. A. J. Hu and M. Y. Vardi, editors. *Proc. 10^{th} Intl. Conference on Computer Aided Verification*, volume 1427 of *LNCS*. Springer-Verlag, June 1998.
33. Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In J. W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rosenberg, editors, *Proceedings of the REX Workshop "A Decade of Concurrency: Reflections and Perspectives"*, volume 803 of *LNCS*, pages 273–346. Springer-Verlag, 1994.
34. Y. Kesten, Z. Manna, and A. Pnueli. Verifying clocked transition systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *LNCS*, pages 13–40. Springer-Verlag, 1996.
35. R. P. Kurshan. Testing containment of ω-regular languages. Technical Report 1121-861010-33, Bell Labs, 1986.
36. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.
37. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):435–455, 1974.
38. L. Lamport. The synchronization of independent processes. *Acta Informatica*, 7(1):15–34, 1976.
39. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
40. L. Lamport and L. C. Paulson. Should your specification language be typed? Research Report 147, DEC Systems Research Center, Palo Alto, CA, May 1997.
41. 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.
42. Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. S. Chang, M. Colón, L. de Alfaro, H. Devarajan, H. B. Sipma, and T. E. Uribe. STeP: The Stanford temporal prover. Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.
43. Z. Manna, A. Browne, H. B. Sipma, and T. E. Uribe. Visual abstractions for temporal verification. In A. Haeberer, editor, *Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *LNCS*, pages 28–41. Springer-Verlag, Dec. 1998.
44. Z. Manna, M. A. Colón, B. Finkbeiner, H. B. Sipma, and T. E. Uribe. Abstraction and modular verification of infinite-state reactive systems. In M. Broy, editor, *Requirements Targeting Software and Systems Engineering (RTSE)*, LNCS. Springer-Verlag, 1998. To appear.
45. Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.
46. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
47. Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.
48. 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*, volume 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

49. Z. Manna and A. Pnueli. Verification of parameterized programs. In Börger [12], pages 167–230.
50. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
51. 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*, volume 1386 of *LNCS*, pages 305–318. Springer-Verlag, Apr. 1998.
52. Z. Manna and H. B. Sipma. Verification of parameterized systems by dynamic induction on diagrams. In *Proc. 11^{th} Intl. Conference on Computer Aided Verification*, LNCS. Springer-Verlag, 1999. To appear.
53. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Pub., 1993.
54. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, Apr. 1980.
55. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [1], pages 411–414.
56. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Found. of Comp. Sci.*, pages 46–57. IEEE Computer Society Press, 1977.
57. A. Pnueli. Lecture notes: the Bakery algorithm. Draft Manuscript, Weizmann Institute of Science, Israel, May 1996.
58. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Intl. Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.
59. R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, Jan. 1984.
60. H. B. Sipma. *Diagram-based Verification of Discrete, Real-time and Hybrid Systems*. PhD thesis, Computer Science Department, Stanford University, Feb. 1999.
61. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers (North-Holland), 1990.
62. T. E. Uribe. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Computer Science Department, Stanford University, Dec. 1998. Technical Report STAN-CS-TR-99-1618.
63. M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comp. Sys. Sci.*, 32:183–221, 1986.