[illegible]

NASA NAG2-892: Software Development Technologies for Reactive, Real-Time, and Hybrid Systems —Summary of Research—

P.I.: Prof. Zohar Manna
Computer Science Department
Stanford University
Stanford, CA. 94305-9045

Period covered: 10/1/93 — 12/31/98

Contents

1 Introduction

This research is directed towards the implementation of a comprehensive deductive-algorithmic environment (toolkit) for the development and verification of high assurance reactive systems, especially concurrent, real-time, and hybrid systems. For this, we have designed and implemented the STeP (Stanford Temporal Prover) verification system.

Reactive systems have 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 procedures for proving that a given system satisfies a given temporal property.

The research covered necessary theoretical foundations as well as implementation and application issues. We summarize the theoretical results in Section 2, and then describe, in more detail, the implementation and tools that we developed in Section 3.

Reactive, Real-time and Hybrid Systems

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 *finite-state*, since they can be in only finitely many distinct states; the state depends only on 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. 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.

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 processes.

More dimensions of infinity are introduced when considering *real-time systems*, where time advances continuously and the time elapsed between

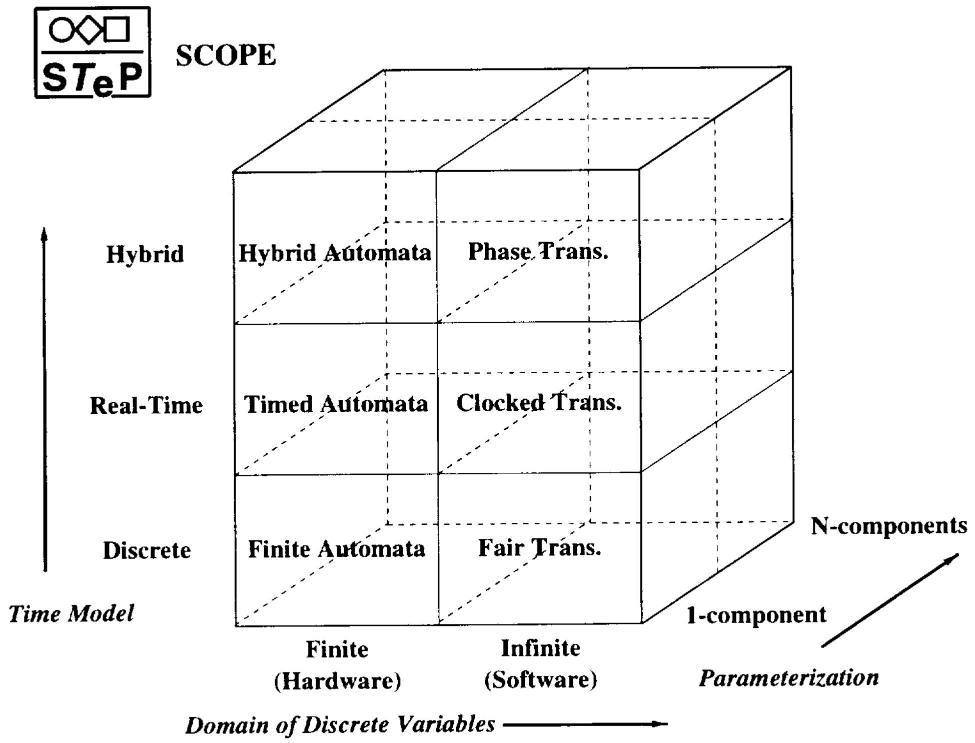


Figure 1: Scope of STeP

events can be measured. A further extension is given by *hybrid systems*, where continuous variables evolve over time as determined by differential equations.

The various systems STeP can verify differ in their time model—discrete, real-time, or hybrid—as well as in the domain of their state variables, which can be finite or infinite. Furthermore, systems can be *parameterized* in the number of processes that compose them (N-process systems). All of these systems can be modeled, however, using the same underlying computational model: (fair) transition systems [MP95]. This basic model is extended in appropriate ways to allow for modular structures, hardware-specific components, clocks, or continuous variables. Figure 1 describes the scope of STeP, classified along these three main dimensions.

2 Theoretical Foundations

2.1 System Representation: Transition Systems

The basic system representation in STeP uses a set of *transitions*. Each transition is a relation over unprimed and primed system variables, expressing the values of the system variables at the current and next state. Transitions can thus be represented as general first-order formulas, though more specialized notations for guarded commands and assignments is also available. In the discrete case, transitions can be labeled as *just* or *compassionate*; such fairness constraints are relevant to the proof of progress properties (see [MP95]).

SPL Programs: For convenience, discrete systems can be described in the Simple Programming Language (SPL) of [MP95]. SPL programs are automatically translated into the corresponding fair transition systems, which are then used as the basis for verification.

Real-Time Systems: STeP can verify properties of real-time systems, using the computational model of *clocked transition systems* [MP96]. Clocked transition systems consist of standard instantaneous transitions that can reset auxiliary clocks, and a *progress condition* that limits the time that the system can stay in a particular discrete state. Clocked transition systems are converted into discrete transition systems by including a tick transition that advances time, constrained by the progress condition. The tick transition is parameterized by a positive real-valued duration of the time step.

Hybrid Systems: *Hybrid transition systems* generalize clocked transition systems, by allowing real-valued variables other than clocks to vary continuously over time. The evolution of continuous variables is described by a set of constraints, which can be in the form of sets of differential equations or differential inclusions. Similar to clocked transition systems, hybrid transition systems are converted into discrete transition systems by including a tick transition, parameterized by the duration of the time step. However, for hybrid systems the tick transition must not only update the values of the clocks, which is straightforward, but must also determine the value of the continuous variables at the end of the time step. The updated value of the continuous variables is represented symbolically; axioms and invariants, generated based on the constraints, are used to determine the actual value or the range of values at the time they are needed.

Other formalisms such as timed transition systems, timed automata and hybrid automata can be easily translated into hybrid and clocked transition systems [MP96].

Modularity: Complex systems are built from smaller components. Most modern programming languages and hardware description languages therefore provide the concept of *modularity*. STeP includes facilities for modular specification and verification [FMS98], based on *modular transition systems*, which can concisely describe complex transition systems. Each module has an *interface* that determines the observability of module variables and transitions. The interface may also include an *environment assumption*, a relation over primed and unprimed interface variables that limits the possible environments the module can be placed in. The module can only be composed with other modules that satisfy the environment assumption. Communication between a module and its environment can be asynchronous, through shared variables, and synchronous, through synchronization of labeled transitions.

More complex modules can be constructed from simpler ones by possibly recursive module expressions, allowing the description of hierarchical systems of unbounded depth. Module expressions can refer to modules defined earlier, or instances of parameterized modules, enabling the reuse of code and of properties proven about these modules. Besides the usual hiding and renaming operations, the language provides a construct to augment the interface with new variables that provide a summary value of multiple variables within the module. Symmetrically, a restriction operation allows the module environment to combine or rearrange the variables it presents to the module.

Real-time and hybrid systems can also be described as modular systems; discrete, real-time and hybrid modules may be combined into one system. The evolution constraints of hybrid modules may refer to continuous variables of other modules, thus enabling the decomposition of systems into smaller modules. To enable proofs of nontrivial properties over such modules, we allow arbitrary constraints on these external continuous variables in the environment assumption.

2.2 Property Specification: Temporal Logic

We use *linear-time temporal logic* (LTL) as our property specification language. Formulas of LTL describe sets of infinite sequences of states. We say that a system S satisfies a temporal property φ, written $S \models \varphi$, if every computation of S satisfies φ.

The temporal logic is defined relative to an *assertion language*, which is used to characterize sets of states. For this, we use the full expressive power of first-order logic, including both interpreted function symbols and predicates. This logic is supported by the corresponding automated deductive (theorem-proving) tools.

The system models of clocked and hybrid transition systems (see above) do not require any extension to the property specification language; the global clock is a system variable that can be directly referenced in temporal specifications. The assertion language can describe constraints on the clocks and, in the case of hybrid systems, other continuous variables.

2.3 Deductive Verification

The deductive methods of STeP verify temporal properties of systems by means of verification rules and verification diagrams. *Verification rules* reduce temporal properties of systems to first-order verification conditions [MP95]. *Verification diagrams* [MP94] provide a visual language for guiding, organizing, and displaying proofs, and automatically generating the appropriate verification conditions as well (see Section 3.3).

Since clocked and hybrid transition systems are converted into fair transition systems, verification rules and diagrams are uniformly applicable to discrete, real-time and hybrid systems. However, due to the parameterization of the tick transition, the resulting verification conditions for real-time and hybrid systems are usually more complex than those for (unparameterized) discrete systems.

2.4 Deductive-Algorithmic Verification

Algorithmic methods such as *model checking* can automatically verify temporal properties of reactive systems, but are restricted to finite-state systems (or very specialized classes of infinite-state ones). We have developed a number of formalisms that combine deductive methods and algorithmic methods to verify, more automatically, general infinite-state systems, and extending the expressiveness of model checking tools.

2.4.1 Deductive Model Checking

Deductive model checking [SUM99] allows the interactive model checking of infinite-state systems. Standard explicit-state model checking searches the product of system's state-space and the tableau (automaton) for the negation of the temporal property being verified, in the search for a counterexample computation. Deductive model checking transforms a diagram that abstracts this product, called a *falsification diagram*, starting with a general skeleton of the product graph and refining it until a counterexample is found, or the impossibility of such a counterexample is demonstrated.

The deductive model checking (DMC) procedure starts with an initial falsification diagram that embeds all models of the negation of the property. Transformations are then applied, producing a sequence of falsification diagrams. Each transformation preserves the computations of the system that are embedded in the diagram, guaranteeing that each falsification diagram includes all system computations that violate the property. If we obtain a diagram that does not embed any computation, then the system satisfies the property.

In the general infinite-state case, the deductive model checking procedure will be interactive, and is not guaranteed to terminate. In the finite-state case, it can be used as a decision procedure for establishing temporal properties, as done by standard model checking. However, the entire state-space does not always have to be explored in this case.

2.4.2 Generalized Verification Diagrams

Verification Diagrams provide a graphical representation of a deductive proof, summarizing the necessary verification conditions, and are therefore easier to construct and understand. *Generalized Verification Diagrams* extend them to be applicable to arbitrary temporal properties, replacing the well-formedness check on the diagram by a finite-state model checking step. They are a *complete* proof method for general (state-quantified) temporal formulas, relative to the reasoning required to establish verification conditions.

The diagrams of [BMS95] use *Street acceptance conditions*. In [MBSU98, Sip98], we present an alternative description of generalized verification diagrams based on *Müller acceptance conditions*, which are less concise but more intuitive to the user.

The thesis [Sip98] presents diagram-based formalisms to verify temporal properties of reactive system. Generalized verification diagrams represent the temporal structure of the program as relevant to the property they prove. The deductive component of a verification diagram defines a set of first-order verification conditions that, when proven valid, show that all behaviors of the system are embedded in the diagram. The algorithmic component is an automata-theoretic language inclusion check that determines whether all behaviors of the diagram satisfy the property.

We show how these methods can be used to verify not only discrete systems, but real-time and hybrid systems as well. We also present two specialized classes of diagrams for these systems: *nonzenoness diagrams* represent a proof that a real-time or hybrid system is time-divergent, that is, all behaviors of the system can be extended into behaviors in which time grows beyond any bound. *Receptiveness diagrams* prove a related property of real-time and hybrid modules that implies time divergence and is preserved by parallel composition.

2.4.3 Abstraction

In [CU98], we present an algorithm that uses decision procedures to generate finite-state abstractions of possibly infinite-state systems. The algorithm compositionally abstracts the transitions of the system, relative to a given, fixed set of assertions. Thus, the number of validity checks is proportional to the size of the system description, rather than the size of the abstract state-space. The generated abstractions are weakly preserving for $\forall$CTL* temporal properties, including LTL.

The thesis [Uri98] presents an abstraction-based framework for verifying temporal properties of reactive systems, to allow more automatic verification of general infinite-state systems and the verification of larger finite-state ones. Underlying these deductive-algorithmic methods is the theory of *property-preserving assertion-based abstractions*, where a finite-state abstraction of the system is deductively justified and algorithmically model checked.

3 Implementation: STeP

The Stanford Temporal Prover (STeP) is a tool for the computer-aided formal verification of reactive systems, including real-time and hybrid systems, based on their temporal specification. STeP integrates model checking and deductive methods to allow the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains.

Figure 2 presents an outline of the STeP system. The main inputs are a reactive system and a property to be proven for it, expressed as a temporal logic formula. The system can be a hardware or software description, and include real-time and hybrid components. Verification is performed by model checking or deductive means or a combination of the two.

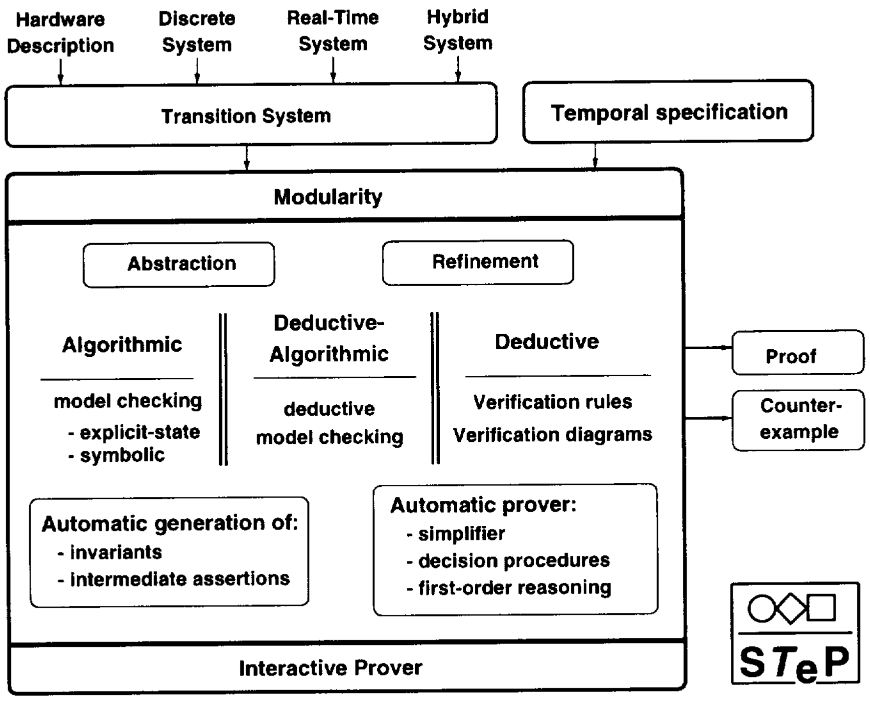


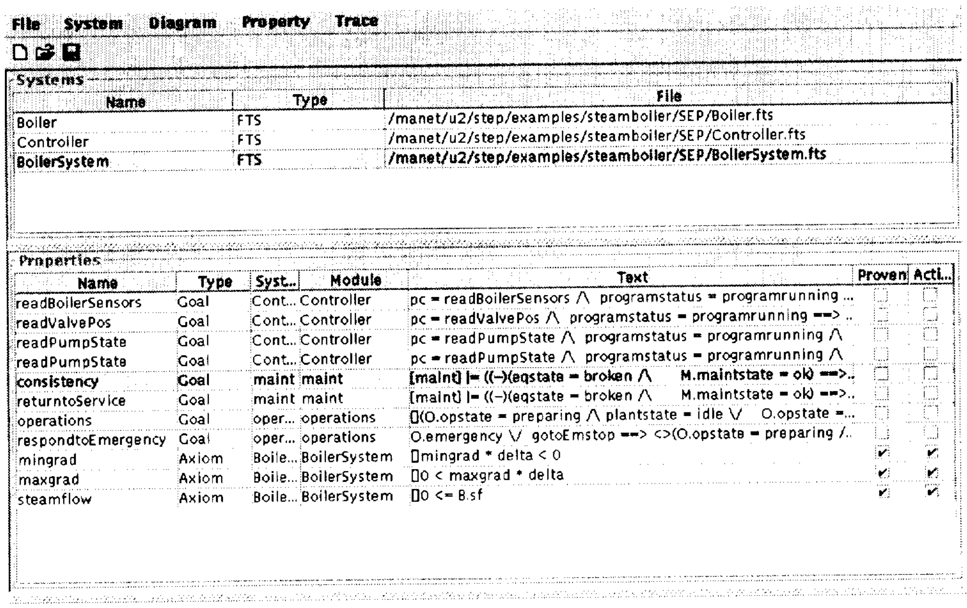
Figure 2: An outline of the STeP system

3.1 User Interface

STeP 2.0 has a graphical user interface implemented in Java. The deductive verification and theorem-proving components of STeP are implemented in Standard ML of New Jersey, while the model checking tools are implemented in C, for added efficiency. The ML component of STeP can also access external OBDD libraries implemented in C.

STeP provides a comprehensive, integrated environment to prove temporal properties over reactive systems. The STeP Session Editor, presented in Figure 3, keeps track of the main properties of interest throughout the verification session, including axioms, assumptions, previously proven properties, and automatically generated invariants, as well as the module to which each applies. Thus, it can handle multiple systems and proofs simultaneously. Properties can be activated or deactivated to control the extent of their use in automatic theorem-proving.

Figure 4 shows the STeP Proof Editor, which is used to apply the basic deductive temporal verification rules as well as the Gentzen-style interactive



Systems

Name	Type	File
Boiler	FTS	/manet/u2/step/examples/steamboiler/SEP/Boiler.fts
Controller	FTS	/manet/u2/step/examples/steamboiler/SEP/Controller.fts
BoilerSystem	**FTS**	**/manet/u2/step/examples/steamboiler/SEP/BoilerSystem.fts**

Properties

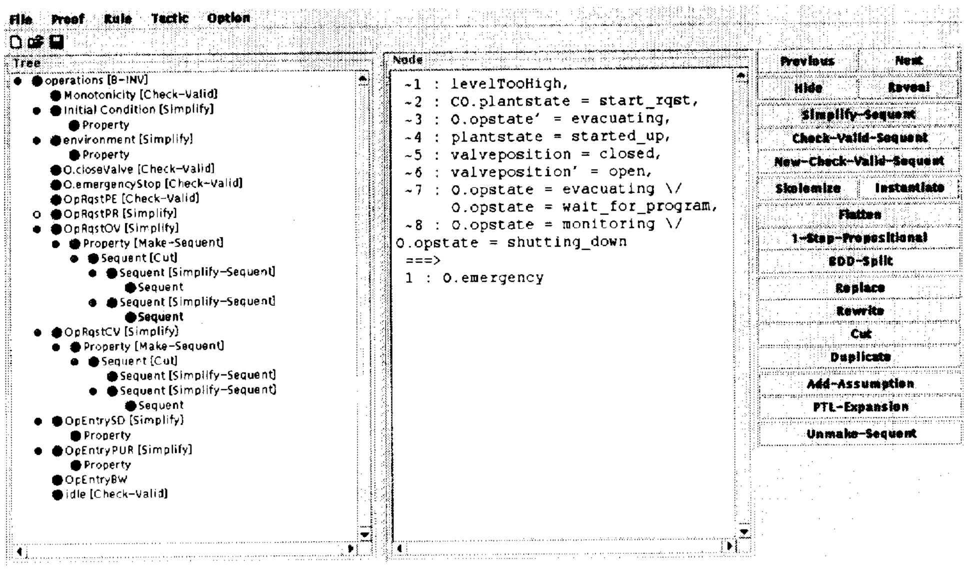
Name	Type	Syst...	Module	Text	Proven	Acti...
readBoilerSensors	Goal	Cont...	Controller	pc = readBoilerSensors /\ programstatus = programrunning ...	☐	☐
readValvePos	Goal	Cont...	Controller	pc = readValvePos /\ programstatus = programrunning ==> ..	☐	☐
readPumpState	Goal	Cont...	Controller	pc = readPumpState /\ programstatus = programrunning /\	☐	☐
readPumpState	Goal	Cont...	Controller	pc = readPumpState /\ programstatus = programrunning /\	☐	☐
consistency	**Goal**	**maint**	**maint**	**[maint] |= ((-)(eqstate = broken /\ M.maintstate = ok) ==>..**	☐	☐
returntoService	Goal	maint	maint	[maint] |= ((-)(eqstate = broken /\ M.maintstate = ok) ==>..	☐	☐
operations	Goal	oper...	operations	□(O.opstate = preparing /\ plantstate = idle \/ O.opstate =...	☐	☐
respondtoEmergency	Goal	oper...	operations	O.emergency \/ gotoEmstop ==> <>(O.opstate = preparing /..	☐	☐
mingrad	Axiom	Boile...	BoilerSystem	□mingrad * delta < 0	✓	✓
maxgrad	Axiom	Boile...	BoilerSystem	□0 < maxgrad * delta	✓	✓
steamflow	Axiom	Boile...	BoilerSystem	□0 <= B.sf	✓	✓

Figure 3: STeP Session Editor

theorem proving rules. In a typical deductive verification effort, the top-level goal is a temporal formula to be proven valid for a given system. Verification rules or diagrams are used to generate verification conditions, as subgoals, which together imply the system validity of the original temporal property. These subgoals are then established automatically using decision procedures (Section 3.6) or interactively using the Gentzen-style rules. Model checking is also initiated by the Proof Editor.

3.2 Model Checking

STeP features automatic explicit-state and symbolic model checking for linear-time temporal logic. The *explicit-state model checker* performs an incremental (depth-first) search of the state-space, directed by the temporal tableau (automaton) for the negated specification. Thus, only those states that can potentially violate the specification are visited. This enables the use of the explicit-state model checker on some infinite-state systems, though it is not guaranteed to terminate for these systems. The *symbolic model checker* uses a breadth-first search through sets of states represented



```
● ●operations [B-INV]
    ●Monotonicity [Check-Valid]
  ● ●Initial Condition [Simplify]
      ●Property
  ● ●environment [Simplify]
      ●Property
    ●O.closeValve [Check-Valid]
    ●O.emergencyStop [Check-Valid]
    ●OpRqstPE [Check-Valid]
  o ●OpRqstPR [Simplify]
  ● ●OpRqstOV [Simplify]
    ● ●Property [Make-Sequent]
      ● ●Sequent [Cut]
        ● ●Sequent [Simplify-Sequent]
            ●Sequent
        ● ●Sequent [Simplify-Sequent]
            ●Sequent
  ● ●OpRqstCV [Simplify]
    ● ●Property [Make-Sequent]
      ● ●Sequent [Cut]
          ●Sequent [Simplify-Sequent]
        ● ●Sequent [Simplify-Sequent]
            ●Sequent
  ● ●OpEntrySD [Simplify]
      ●Property
  ● ●OpEntryPUR [Simplify]
      ●Property
    ●OpEntryBW
    ●idle [Check-Valid]
```

```
 ~1 : levelTooHigh,
 ~2 : CO.plantstate = start_rqst,
 ~3 : O.opstate' = evacuating,
 ~4 : plantstate = started_up,
 ~5 : valveposition = closed,
 ~6 : valveposition' = open,
 ~7 : O.opstate = evacuating \/
      O.opstate = wait_for_program,
 ~8 : O.opstate = monitoring \/
O.opstate = shutting_down
 ===>
 1 : O.emergency
```

Figure 4: STeP Proof Editor

by ordered binary decision diagrams (OBDDs). Thus, it is limited to finite-state systems, whose variables range over a fixed, finite number of values.

When transitions can be expressed as guarded commands (i.e., the system is a set of deterministic actions), symbolic model checking is optimized using techniques for computing predecessor states without computing the entire transition relation. A specialized backwards search for proving invariants is also available. The set of states visited in the backwards search is constrained by auxiliary invariants, which may have been formulated and verified before, or generated automatically.

The symbolic and explicit-state model checkers complement each other. Although limited to finite-state systems, the symbolic model checker can be considerably more efficient, particularly when the state-space is large and the transition relation and fixed points are amenable to representation by OBDD's. On the other hand, the explicit-state model checker is often faster on systems with relatively few reachable states.

3.3 Generalized Verification Diagrams

Generalized verification diagrams [BMS95, MBSU98] are an extension of

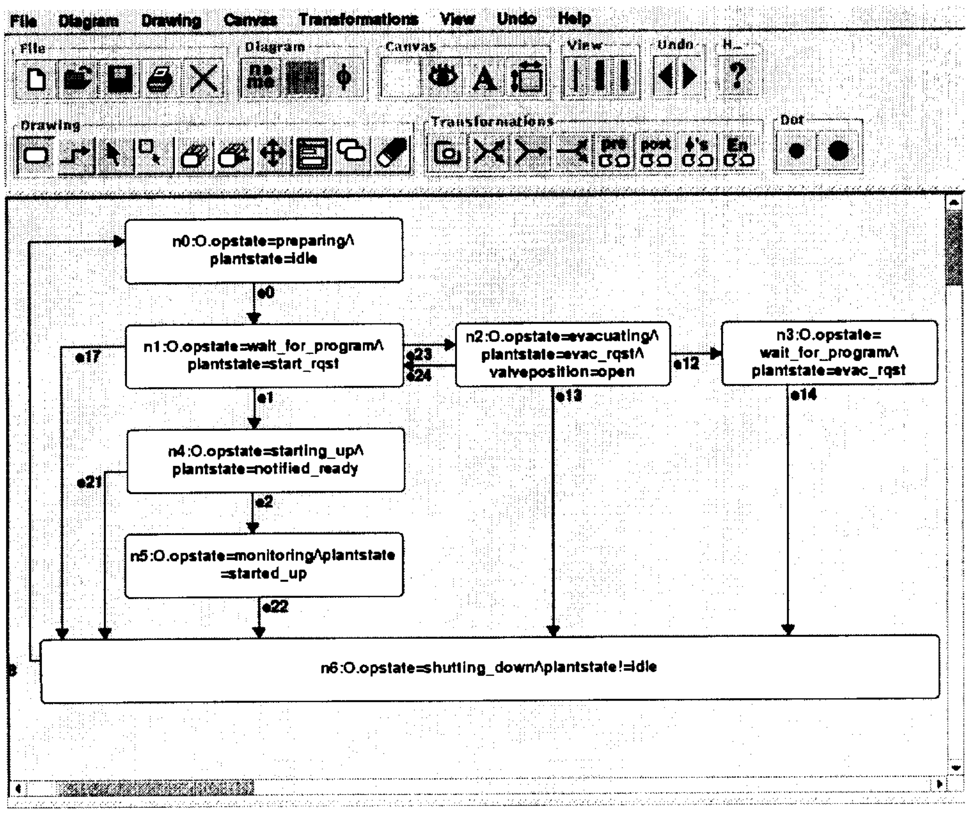

File Diagram Drawing Canvas Transformations View Undo Help
File
Diagram
Canvas
View
Undo
H...
Drawing
Transformations
Dot
pre
post
ϕ's
En
n0:O.opstate=preparing∧ plantstate=idle
e0
n1:O.opstate=wait_for_program∧ plantstate=start_rqst
e23
e24
n2:O.opstate=evacuating∧ plantstate=evac_rqst∧ valveposition=open
e12
n3:O.opstate= wait_for_program∧ plantstate=evac_rqst
e17
e1
e13
e14
n4:O.opstate=starting_up∧ plantstate=notified_ready
e21
e2
n5:O.opstate=monitoring∧plantstate =started_up
e22
n6:O.opstate=shutting_down∧plantstate!=idle

Figure 5: STeP Diagram Editor

verification diagrams that allow the verification of arbitrary temporal properties. Diagrams can be seen as intermediaries between the system and the property to be proven. A set of verification conditions is proved, deductively, to show that the diagram faithfully represents computations of the system. An algorithmic check then establishes that the diagram corresponds to the formula being proved. Together, these two stages show that all computations of the system are models of the temporal property.

The STeP Diagram Editor, shown in Figure 5, allows the user to draw a diagram and then prove, using the Proof Editor, the associated verification conditions. In STeP 2.0, the Diagram Editor and the Proof Editor are more tightly coupled, to facilitate the incremental development of diagrams. The user can draw an initial version and try to prove the associated verification conditions. If they fail, the user can make local corrections to the diagram (or discover something wrong with the system) and attempt the proof again.

The verification conditions are *local* to the diagram; failed verification conditions point to missing edges or nodes, weak assertions, or possible bugs in the system. Since local changes to a diagram do not affect the verification conditions elsewhere, much of the work from the previous iteration can be saved. Using feedback from the Proof Editor, the Diagram Editor can highlight proved and unproved edges and nodes in the diagram, helping the user correct the diagram. A change to the diagram automatically invalidates the verification conditions in the Proof Editor that are affected by the change.

3.4 Constructing Finite-State Abstractions

Temporal properties can be proved for a complex system by finding a simpler *abstract system* such that if the abstract system satisfies a related property, then the original *concrete system* satisfies the original one as well. If the abstract system is finite-state, its temporal properties can be established automatically using a model checker. We have developed methods for automatically generating finite-state abstractions of possibly infinite-state systems, using the decision procedures in STeP [CU98, Uri98]. We describe some of these decision procedures in Section 3.6.

The abstraction algorithm compositionally abstracts the transitions of the system, expressed as first-order relations, relative to a given, fixed set of assertions which define the abstract state-space. The number of validity checks is proportional to the size of the system description, rather than the size of the abstract state-space.

Once the finite-state abstraction is generated, it can be model checked, explicitly or symbolically. The generated abstractions are *weakly preserving* for universal ($\forall$CTL*) temporal properties, including LTL. This means that validity at the abstract level implies the validity of the original property over the concrete system; however, if the abstract property fails, the original property might still hold. In this case, we say that the abstraction was not fine enough. An abstract counterexample can be used, manually, to determine if a corresponding concrete counterexample exists, or else to build a finer abstraction.

3.5 Automatic Invariant Generation

Deductive 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. STeP implements techniques for the *automatic generation of invariants*, as described in [BBM97]. Invariant generation is based on approximate propagation, starting from the set of initial states, through the state-space of the system until a fixpoint is reached. Depending on the approximation method used, different types of invariants can be generated:

- *Local invariants* result from analyzing the possible values of individual variables, as well as the relation between control locations and data values.
- *Linear invariants* express linear relationships between system variables.
- *Polyhedral invariants* generalize linear invariants, expressing polyhedral constraints over sets of system variables.

For real-time and hybrid systems STeP provides an alternative technique of invariant generation, also based on forward propagation of system behavior through the state space, but now starting from the entire state space [BMSU98]. In this case every propagation step leads to an invariant; no fixpoint needs to be computed. For hybrid systems these techniques have been further optimized to take advantage of the structure of the constraints, resulting in stronger invariants. In [MS98] we show an example where the invariants thus generated are sufficiently strong to prove the main property of interest.

3.6 Decision Procedures

The verification conditions generated in deductive verification refer to the domain of computation of the system being verified. To establish verification conditions in the most automatic and efficient manner, STeP includes *decision procedures* for a number of theories frequently used in computation domains, and thus common in formal verification [Bjø98].

The basic integration of decision procedures is a variant of Shostak's congruence closure-based algorithm. The version in STeP admits integration with special relations such as ordering constraints, non-convex theories, and cyclic data-structures. At the top-level, an algorithm based on congruence closure propagates equality constraints through function symbols. It invokes the other decision procedures as auxiliary simplifiers and solvers. The theories supported in this way include:

- *Partial orders.* Beyond basic equality, partial orders are a more expressive constraint language to specify relations between variables. Each partial order constraint is represented as an edge in a graph whose nodes are congruence closure equivalence classes.

- *Linear and non-linear arithmetic.* STeP provides Fourier's quantifier elimination procedure to deal with formulas involving linear arithmetic; this procedure also extracts implied equalities. Verification conditions involving nonlinear arithmetic, which are common in the verification of hybrid systems, are dealt with by techniques that eliminate first- and second-degree variables. Many verification conditions involving arithmetical symbols are restricted to linear arithmetic, where multiplication is only used with at most one non-numerical argument. For this, STeP includes Fourier's quantifier elimination procedure, which also extracts implied equalities.

- *Bit-vectors.* Reasoning about bit-vectors is essential for hardware verification. STeP includes decision procedures for fixed-size bit-vectors with boolean bitwise operations and concatenation, and for non-fixed size bit-vectors with concatenation [BP98].

- *Lists, queues,* and *word decision procedures.* Lists and queues are common data structures, especially in systems using abstract datatypes or asynchronous channels. Both lists and queues can be viewed as special cases of words, with concatenation being the basic operation. Although the known decision procedures for word equalities have prohibitive complexity, the special cases of lists and queues can be solved efficiently.

- *Recursive data-types.* STeP supports equality reasoning for general recursive datatypes, which allow the specification of S-expressions and other tree-like structures. Enumeration types and records are treated as special cases of recursive datatypes.

  Co-inductive data-types, such as lazy lists, are also supported. Both equality constraints and subterm relationships are supported in the integration of decision procedures.

- *Set theory.* STeP provides basic support for Multi-level Syllogistic Set-theory (MLSS). MLSS terms range over sets, and operations include union, intersection, set-difference, and finite set-enumeration. Atomic relations include set equality, inclusion and membership.

STeP uses decision procedures not only to check validity, but to *simplify* formulas as well, rewriting them to smaller, logically equivalent ones. Efficient formula simplification can make verification conditions more readable and manageable, and improves the efficiency of subsequent validity checking.

Validity Checking: The above decision procedures check validity of *ground* formulas, where no first-order quantification is present. STeP extends this combination of ground decision procedures to include theory-specific unification algorithms, which find quantifier instantiations needed for first-order validity checking [BSU97].

The STeP validity checker is an extension of the Davis-Putnam-Loveland-Logemann propositional satisfiability checker. It operates on formulas in nonclausal form, and is extended to consider quantifiers. The procedure is intended to preserve the original structure of the formula, including structure sharing using `let`- expressions, as much as possible. Case splitting, instantiation, skolemization and simplification can all be performed incrementally, in a uniform setting. The procedure takes advantage of instantiations suggested by decision procedures whenever available, but can also use "black-box" procedures that only provide yes/no answers.

Interactive Theorem Proving: An interactive Gentzen-style theorem prover is available as part of the Proof Editor to establish verification conditions that are not proved automatically. This prover features induction over well-founded domains, and Gentzen rules for temporal reasoning. It is complete for (uninterpreted) first-order logic: if the formula is valid, a proof for it can be found.

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, the use of *cut formulas*, which are arbitrary user-provided formulas used to perform case analysis, and the duplication and instantiation of quantified formulas.

3.7 Modular Verification

Different components of a large system may require the application of different verification methodologies, depending on their specific type (real-time or discrete, finite- or infinite-state). Using the notion of modular validity, modular properties can be established by the same set of methods as global properties, accounting for environment transitions. Automatic property inheritance then ensures that such properties can be used as lemmas in proofs over composite modules. In the case of recursively defined systems, properties can be established by structural induction.

Many properties are not directly guaranteed by a module, but hold only under certain assumptions. STeP's proof management allows assumptions to be used before their proof is available, checking the resulting dependency diagram to avoid unsound circular reasoning. Assumptions about the environment can be made when proving a modular property, and subsequently discharged when the module is composed with another. The search for appropriate assumptions can be guided by constructing verification diagrams for each module and attempting to prove the associated verification conditions [FMS98, MCF$^+$98].

4 Applications

4.1 Real-time Systems

In [BMSU97], we present a modular framework for proving temporal properties of real-time systems, based on clocked transition systems and linear-time temporal logic. We show how deductive verification rules, verification diagrams, and automatic invariant generation can be used to establish properties of real-time systems in this framework. We also discuss global and modular proofs of the branching-time property of non-Zenoness. As an example, we present the mechanical verification of the *generalized railroad crossing* case study using STeP.

4.2 Fault-tolerant Systems

In [BLM97], a parameterized fault-tolerant leader-election algorithm recently proposed is modeled and verified using STeP. Our methods settle the general N-process correctness for the algorithm, which had been previously verified only for $N = 3$. We formulate the notion of *Uniform Compassion* to model progress in faulty systems more faithfully, and combine it with the more standard notions of fairness. We also show how the correctness proofs generalize to different channel models by a reduction to a simple channel model.

4.3 Hybrid Systems

In [MS98] we present invariant generation methods for hybrid systems, and verify a simple hybrid water level controller using STeP. In [MCF+98], we show how deductive verification tools, and the combination of finite-state model checking and abstraction, allow the verification of infinite-state systems featuring data types commonly used in software specifications, including real-time and hybrid systems. We verify properties of the hybrid industrial steam boiler example, which is modelled as the combination of discrete and hybrid modules.

4.4 Educational Use

STeP has been used on several occasions for teaching a graduate-level introductory course on temporal verification of reactive systems, at Stanford University and at the Technion (Israel Institute of Technology, Haifa).

STeP is freely available for research and educational use. The STeP manual is available as [BBC+95], and system descriptions are provided in [BBC+96, MBB+98]. For information on obtaining STeP, see the STeP web pages at:

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

5 Graduated Ph.D. Students

(Partially supported by this contract.)

1. Arjun Kapur (1997). Thesis: *Interval and Point-based Approaches to Hybrid System Verification.*
2. Luca de Alfaro (1997). Thesis: *Formal Verification of Probabilistic Systems.*
3. Nikolaj Bjorner (1998). Thesis: *Integrating Decision Procedures for Temporal Verification.*
4. Tomás E. Uribe (1998). Thesis: *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems.*
5. Henny B. Sipma (1999). Thesis: *Diagram-based Verification of Discrete, Real-time and Hybrid Systems.*

References

[BBC$^+$95] Nikolaj S. Bjørner, Anca Browne, Eddie S. Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[BBC$^+$96] Nikolaj S. Bjørner, Anca Browne, Eddie S. Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. 8th Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 415–418. Springer-Verlag, July 1996.

[BBM97] Nikolaj S. Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 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.

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

[BLM97] Nikolaj S. Bjørner, Uri Lerner, and Zohar Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Intl. Conf. on Temporal Logic*. Kluwer, 1997. To appear.

[BMS95] Anca Browne, Zohar Manna, and Henny 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.

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

[BMSU98] Nikolaj S. Bjørner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Deductive verification of real-time systems using STeP. Technical Report STAN-CS-TR-98-1616, Computer Science Department, Stanford University, January 1998. To appear in *Theoretical Computer Science*. Preliminary version appeared in *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.

[BP98] Nikolaj S. Bjørner and Mark C. Pichora. Deciding fixed and non-fixed size bit-vectors. In *4th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1384 of *LNCS*, pages 376–392. Springer-Verlag, 1998.

[BSU97] Nikolaj S. Bjørner, Mark E. Stickel, and Tomás 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.

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

[FMS98] Bernd Finkbeiner, Zohar Manna, and Henny B. Sipma. Deductive verification of modular systems. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, COMPOS'97*, volume 1536 of *LNCS*, pages 239–275. Springer-Verlag, December 1998.

[MBB$^+$98] Zohar Manna, Nikolaj S. Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Mark Pichora, Henny B. Sipma, and Tomás E. Uribe. An update on STeP: Deductive-algorithmic verification of reactive systems. In *Tool Support for System Specification, Development and Verification*, pages 87–91. Christian-Albrechts-Universitat, Kiel, June 1998. Full version to appear in LNCS.

[MBSU98] Zohar Manna, Anca Browne, Henny B. Sipma, and Tomás E. Uribe. Visual abstractions for temporal verification. In Armando

Haeberer, editor, *AMAST'98*, volume 1548 of *LNCS*, pages 28–41. Springer-Verlag, December 1998.

[MCF+98] Zohar Manna, Michael A. Colón, Bernd Finkbeiner, Henny B. Sipma, and Tomás E. Uribe. Abstraction and modular verification of infinite-state reactive systems. In Manfred Broy, editor, *Requirements Targeting Software and Systems Engineering (RTSE)*, LNCS. Springer-Verlag, 1998. To appear.

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

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

[MP96] Zohar Manna and Amir Pnueli. Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Computer Science Department, Stanford University, April 1996.

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

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

[SUM99] Henny B. Sipma, Tomás E. Uribe, and Zohar Manna. Deductive model checking. To appear in *Formal Methods in System Design*, 1999. Preliminary version appeared in *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, Springer-Verlag, pp. 208–219, 1996.

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