In Rudolf Berghammer and Yassine Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 174–188. Springer-Verlag, 1999.

An Update on STeP: Deductive-Algorithmic Verification of Reactive Systems [⋆]

Zohar Manna,
Nikolaj S. Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner,
Mark Pichora, Henny B. Sipma, and Tomás E. Uribe

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

Abstract. 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 methods for deductive and algorithmic verification, including model checking, theorem proving, automatic invariant generation, abstraction and modular reasoning. We describe the most recent version of STeP, Version 2.0.

1 Introduction

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 1 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 (Section 2). Verification is performed by model checking or deductive means (Section 3), or a combination of the two (Section 4).

This paper presents an overview of the main features of STeP, including recent research and implementation leading up to the latest release, Version 2.0. Earlier descriptions of STeP are available as [BBC$^+$96, BBC$^+$95, MAB$^+$94]. Recent test cases are reported in [BLM97, BMSU98, MS98]. This paper focuses on

⋆ This research was supported in part by the National Science Foundation under grants CCR-95-27927 and CCR-98-04100, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, ARO under grants DAAH04-95-1-0317, DAAH04-96-1-0122 and DAAG55-98-1-0471, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

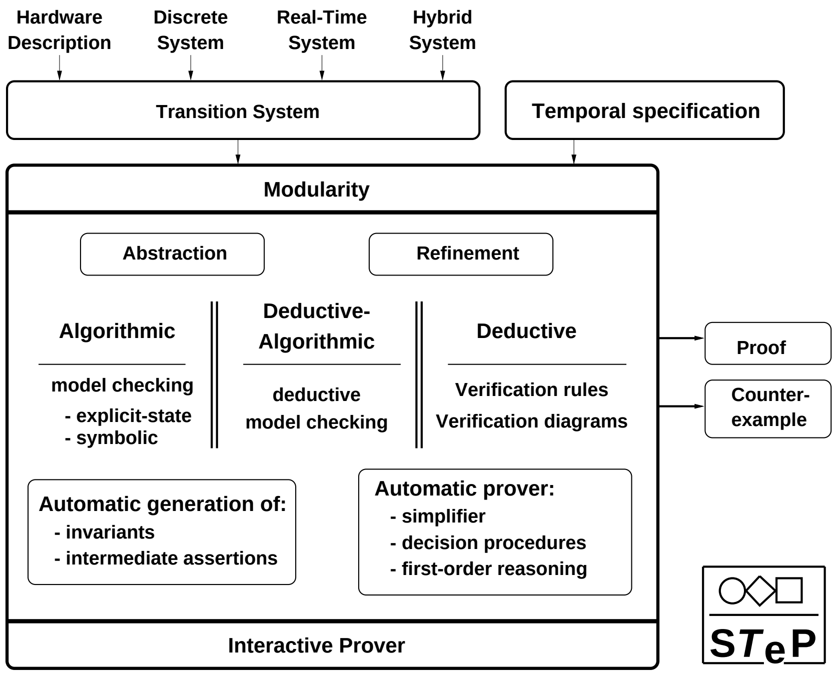


Fig. 1. An outline of the STeP system

the new developments in STeP 2.0, including modular system description and verification, symbolic LTL model checking, generalized verification diagrams, generation of finite-state abstractions, new decision procedures, and a new graphical user interface.

2 Describing Reactive Systems

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 2 describes the scope of STeP, classified along these three main dimensions.

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.

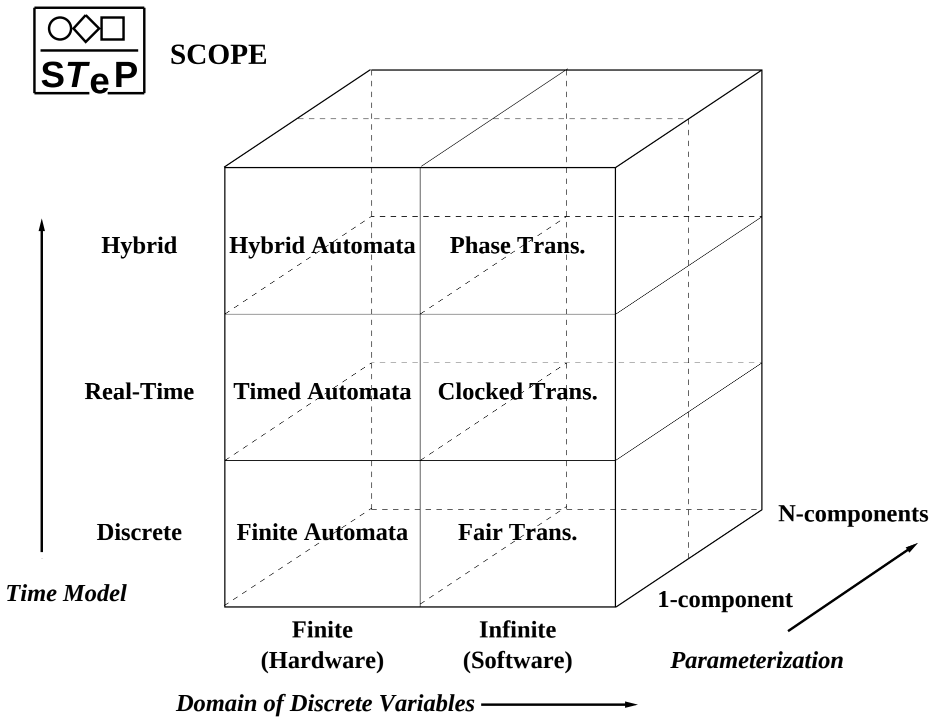


Fig. 2. Scope of STeP

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.

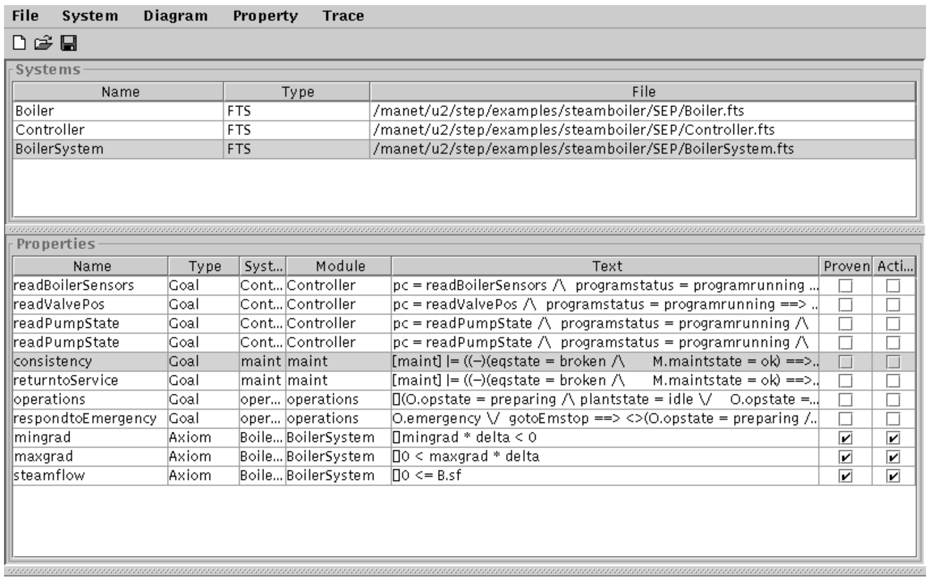
Other formalisms such as timed transition systems, timed automata and hybrid automata can be easily translated into hybrid and clocked transition systems [MP96].

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.

Hardware Description: A Verilog hardware description language front-end has recently been added to STeP. Its main component is a compiler that takes Verilog input and produces a fair transition system, which can then be analyzed



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	☑	☑

Fig. 3. STeP Session Editor

using the deductive and algorithmic tools of STeP.

The goal of this compiler is to produce a faithful representation of the input program, taking into account the delays and events that are part of the Verilog semantics. The compiler extends the Verilog language by allowing parameters to be left unspecified. These parameters can be used to declare bit vectors of arbitrary size, or to compose an array of lower-level modules. These features cater to the deductive component of STeP, which can verify properties of general infinite-state systems.

3 Deductive Verification and Model Checking

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.

3.1 Property Specification: Linear-Time Temporal Logic

We use *linear-time temporal logic* (LTL) to represent properties of reactive systems [MP95]. A model of LTL is an infinite sequence of states. We use the usual temporal operators, such as $\Box p$ (p is always true), $\Diamond p$ (p is eventually true), $p \, \mathcal{U} \, q$ (p is true until q is true, which eventually happens), and $p \, \mathcal{W} \, q$ (p awaits q—p is true at least until q is true, but q need not eventually happen).

We distinguish between *safety* and *progress* properties. Informally, safety properties say that certain "bad states" will never be reached, e.g. as in an invariance $\Box p$ for an assertion p.

Progress properties, on the other hand, can say that "good" states will eventually be reached (perhaps recurrently). Safety properties do not depend on the fairness constraints of the system, whereas progress properties require the justice or compassion of particular transitions in order to be proved.

To specify properties of real-time and hybrid systems, temporal-logic properties can refer to the global and auxiliary clocks, and to the continuous variables; the underlying temporal logic remains the same.

3.2 Deductive Verification

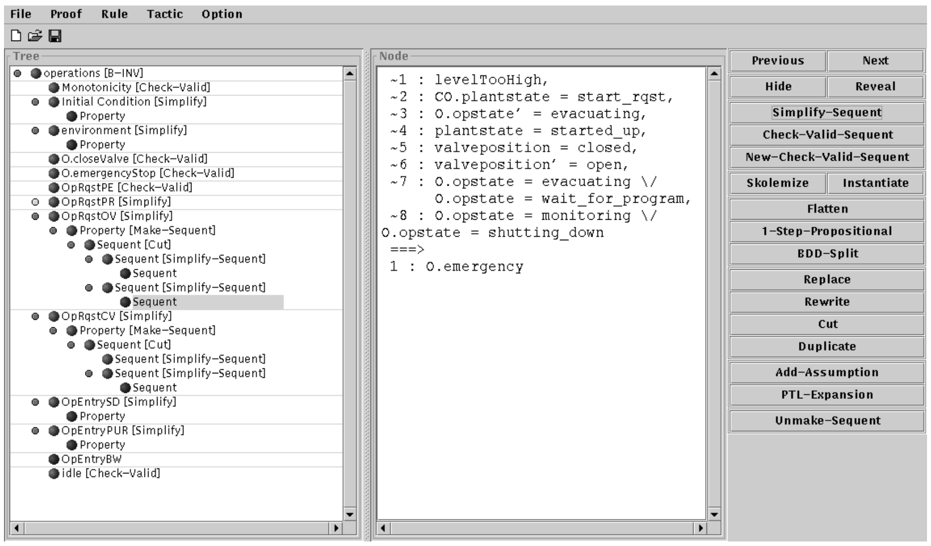
The deductive methods of STeP verify temporal properties of systems by means of verification rules and verification diagrams. *Verification rules* reduce temporal properties of systems to first-order verification conditions [MP95]. *Verification diagrams* [MP94] provide a visual language for guiding, organizing, and displaying proofs, and automatically generating the appropriate verification conditions as well (see Section 4.1).

As 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 transiton, the resulting verification conditions for real-time and hybrid systems are usually more complex than those for (unparameterized) discrete systems.

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 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 5.2) or interactively using the Gentzen-style rules. Model checking is also initiated by the Proof Editor.

3.3 Model Checking

STeP features automatic explicit-state and symbolic model checking for linear-time temporal logic. The *explicit-state model checker* performs an incremental



```
operations [B-INV]
  Monotonicity [Check-Valid]
  Initial Condition [Simplify]
    Property
  environment [Simplify]
    Property
  O.closeValve [Check-Valid]
  O.emergencyStop [Check-Valid]
  OpRqstPE [Check-Valid]
  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
```

Fig. 4. STeP Proof Editor

(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 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 (see Section 5).

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 [McM93]. On the other hand, the explicit-state model checker is often faster on systems with relatively few reachable states.

3.4 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 Combining Deductive and Algorithmic Methods

STeP includes formalisms that combine deductive and algorithmic verification in a number of different ways, which differ in the degree and type of intervention that is required from the user.

4.1 Generalized Verification Diagrams

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

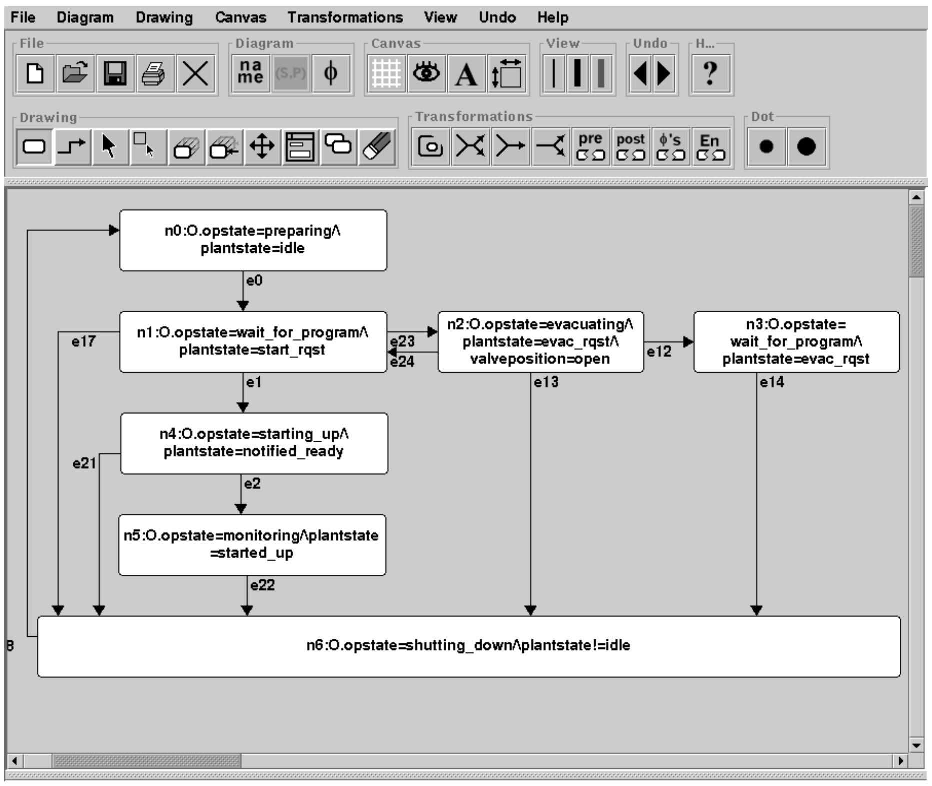

File
Diagram
Drawing
Canvas
Transformations
View
Undo
Help
n0:O.opstate=preparing∧ plantstate=idle
n1:O.opstate=wait_for_program∧ plantstate=start_rqst
n2:O.opstate=evacuating∧ plantstate=evac_rqst∧ valveposition=open
n3:O.opstate= wait_for_program∧ plantstate=evac_rqst
n4:O.opstate=starting_up∧ plantstate=notified_ready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

Fig. 5. STeP Diagram Editor

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.

Deductive model checking [SUM98] uses diagrams to explore and refine the state-space of possibly infinite-state systems, searching for a counterexample computation by transforming the diagram. The STeP Diagram Editor supports some of these diagram transformations for interactive state-space exploration. We will include a more comprehensive implementation in upcoming releases.

4.2 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 5.2.

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 (see Section 3.3). The generated abstractions are *weakly preserving* for universal (∀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.

5 Deductive-Algorithmic Support

Besides model checking, described in Section 3.3, STeP provides two basic automatic tools that support deductive and deductive-algorithmic verification: automatic invariant generation and decision procedures. Both are used extensively in the combinations of deductive and algorithmic verification presented in Section 4.

5.1 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.

5.2 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 [Sho84, CLS96, Bjø98]. 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.
– *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 non-linear arithmetic, which are common in the verification of hybrid systems, are dealt with by techniques that eliminate first- and second-degree variables, as described in [Wei97].
– *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) [CFO89, CZ98]. 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.

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].

As mentioned in Section 3.2, an interactive Gentzen-style theorem prover is available as part of the Proof Editor to establish verification conditions that are not proved automatically.

6 Case Study: Steam boiler

The incorporation of modularity and abstraction in STeP has enabled us to analyze much larger systems than was previously possible. An example is the *steam boiler* case study [ABL96], a benchmark for specification and verification methods for hybrid controlled systems. At the time of its appearance we developed a comprehensive model of this system, including both the plant and the controller. The model consisted of some 1000 lines of SPL code and contained eight parallel processes. However, verification proved impractical and further analysis was suspended. Recently the case study was revived. The system was rewritten as a modular transition system consisting of ten modules with a total of 80 transitions, 18 real-valued variables and 28 finite-domain variables.

Modularity allowed us to prove properties over selected subsystems and inherit them for the full system, thus reducing the number of verification conditions to be proven. In some cases, involving discrete finite-state modules only, the model checker could be applied, making the verification fully automatic. In our previous implementation finite-state components could not be separated from the infinite-state ones, and thus use of the model checker was not possible.

Assertion-based abstraction (see Section 4.2) enabled us to indirectly apply the model checker to infinite-state modules as well, by eliminating the real-valued variables. The relationships between the relevant real-valued variables captured by a small set of assertions were sufficient to let us prove the properties.

A more detailed description of the case study is given in [MCF$^+$98].

7 Implementation

The parsing, theorem-proving and invariant generation components of STeP are implemented in Standard ML of New Jersey. The graphical user interface for STeP 2.0 was developed in Java. The Java graphical packages and the inheritance features of the language are well-suited for the implementation of a variety of visual formalisms with common features, as in the STeP Diagram Editor.

The explicit-state model checker is implemented in C, while the symbolic model checker uses ML linked together with external OBDD libraries, written in C. Similarly, the polyhedral invariant generation uses external polyhedra manipulation routines, implemented in C [HP95].

Stand-alone components: Many of the components of STeP described above can be used in batch mode, as stand-alone components, including:

– the parser/translator from SPL into fair transition systems (Section 2)
– the explicit-state and symbolic model checkers (Section 3.3)
– the linear, local and polyhedral invariant generators (Section 5.1)
– the validity checker and formula simplifier (Section 5.2)

These options are available as command-line options to a separate executable binary file.

External systems: Verification conditions in STeP can be output to external theorem provers or decision procedures. In particular, the MONA package for monadic second order logic can be used; output to the OTTER and Gandalf resolution theorem provers is also provided. STeP interacts with these provers by invoking them in the background and digesting their output to check if the verification conditions have been discharged.

Obtaining STeP

To obtain STeP, send email to `step-request@cs.stanford.edu`. For more information on the system, see the STeP web pages at

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

Acknowledgements: Andrew Miller helped implement the STeP 2.0 interface. Mark Stickel provided AC-unification utilities and feedback. STeP 2.0 uses the OBDD package by Jørn Lind-Nielsen from DTU, Denmark, and the POLKA [HP95] polyhedral manipulation package from VERIMAG.

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

References

[ABL96] J.R. Abrial, E. Boerger, and H. Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, vol. 1165 of *LNCS*. Springer-Verlag, 1996.

[BBC$^+$95] 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, November 1995.

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

[BBM97] N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997. Preliminary version appeared in 1$^{\text{st}}$ *Intl. Conf. on Principles and Practice of Constraint Programming*, vol. 976 of LNCS, pp. 589–623, Springer-Verlag, 1995.

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

[BLM97] 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.

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

[BMSU98] N.S. Bjørner, Z. Manna, H.B. Sipma, and T.E. Uribe. Deductive verification of real-time systems using STeP. Technical report, Computer Science Department, Stanford University, October 1998. Preliminary version appeared in *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.

[BP98] N.S. Bjørner and M.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)*, vol. 1384 of *LNCS*, pages 376–392. Springer-Verlag, 1998.

[BSU97] N.S. Bjørner, M.E. Stickel, and T.E. Uribe. A practical integration of first-order reasoning and decision procedures. In *Proc. of the 14th Intl. Conference on Automated Deduction*, vol. 1249 of *LNCS*, pages 101–115. Springer-Verlag, July 1997.

[CFO89] D. Cantone, A. Ferro, and E. Omodeo. *Computable Set Theory.* Oxford Sceince Publications, 1989.

[CLS96] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In *Proc. of the 13th Intl. Conference on Automated Deduction*, vol. 1104 of *LNCS*, pages 463–477. Springer-Verlag, 1996.

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

[CZ98] D. Cantone and C.G. Zarba. A new fast tableau-based decision procedure for an unquantified fragment of set theory. In *Int. Workshop on First-Order Theorem Proving (FTP'98)*, 1998.

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

[HP95] N. Halbwachs and Y.E. Proy. *POLyhedra desK cAlculator (POLKA).* VERIMAG, Montbonnot, France, September 1995.

[MAB$^+$94] 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.

[MBSU98] Z. Manna, A. Browne, H.B. Sipma, and T.E. Uribe. Visual abstractions for temporal verification. In *AMAST'98*, LNCS. Springer-Verlag, 1998. To appear.

[MCF$^+$98] 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.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Pub., 1993.

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

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

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

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

[Sho84] R.E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, January 1984.

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

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

[Wei97] V. Weispfenning. Quantifier elimination for real algebra—the quadratic case and beyond. In *Applied Algebra and Error-Correcting Codes (AAECC) 8*, pages 85–101, 1997.