In proceedings of *Engineering Automation for Computer Based Systems.*
Carmel October, 1998.

Reactive Verification with Queues

Nikolaj S. Bjørner*†‡§

Abstract

Temporal logic is the *right* specification language to capture requirements of reactive systems. To reason about specifications and implementations, that can in general be infinite state, STeP [3] (The Stanford Temporal Prover) offers verification rules and diagrams to decompose the verification task, generation of invariants to bootstrap verification, and finally integration of decision procedures to dispense with basic verification conditions. We here illustrate the scope of decision procedures for the data-type of queues in verifying a simple reactive controller.

Keywords: Decision procedures, temporal logic requirements, reactive systems.

1 A reactive control problem

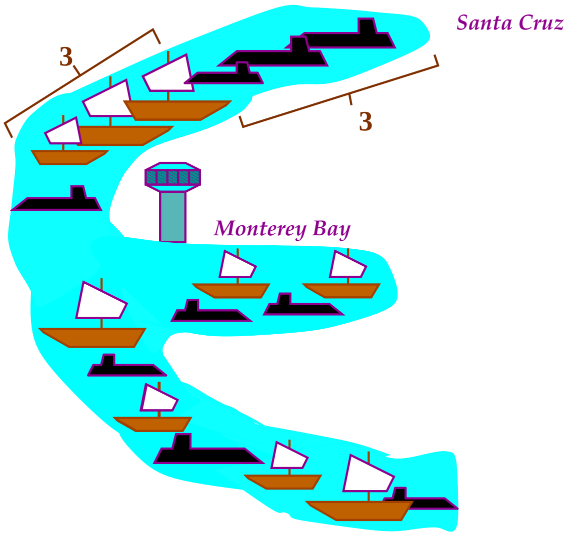


Figure 1: A network traffic controller

To understand the controller that we develop in this paper some familiarity with the Californian geography will be helpful. In particular we will assume that Santa Cruz is north of Monterey Bay. Assume also that the city council in Santa Cruz has decided that the boat traffic passing through Santa Cruz on its way from Monterey should be chunks of equally many submarines and sailboats alternating with a regular period. A frustrated council member asks Clint Eastwood in Carmel to sponsor a tower in Monterey Bay to control the flow of boats, at which Clint asks him, voice rasping, "Are you feeling lucky today, punk?" But that is not our main point. Figure 1 illustrates a possible scenario of the intended flow: boats and submarines from the south enter the bay in any order, the control tower has the option to keep the incoming vessels in the bay for a while or send them up north, but only in such a way that submarines and boats alternate with a regular period of length 3.

As with many other political forums, the council is bitterly divided in Santa Cruz, and is unable to reach a bipartisan agreement of whether the length of the period should be 3 or 5. In the present assembly those favoring a period of length 3 has the majority. Since the council is soon up for re-election it does everything to make sure that popular demand is satisfied, also should the new council have a majority for a period of length 5. A design constraint for the controller is therefore that it should be easily reconfigurable to whatever period N of alternation that fits any assembly.

*This work was supported in part by ARO under grants DAAH04-95-1-0317, the National Science Foundation under grants CCR-95-27927 and CCR-98-04100, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, a gift from Intel, 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).

†E-mail address: `bjorner@kestrel.edu`

‡The author thanks Zohar Manna and the STeP group for a stimulating working environment,

§as well as the organizers of the workshop on *Engineering Automation for Computer Based Systems.*

Since Clint cannot always sit in the control tower we are faced with the problem to design a controller program that directs and redirects boats and submarines to the bay or up north such that the period of alternation is N.

2 Requirements

While pictures may give some intuition as to what sort of controller is to be designed they cannot directly be used to describe the behavior of the controller over time. For this purpose we use linear time temporal logic because it allows to capture requirements from an observer's point of view, directly appeals to state changes over time, and is supported by an adequate supply of model checking algorithms for finite state systems and verification rules for infinite state systems. In linear time temporal logic a formula is evaluated over a computation sequence on the environment's and system's influence on observable variables.

Linear time temporal logic allows for instance to specify that

1. Ships and submarines *do not* alternate within distance N. Thus, for every sequence of at most N vessels sent up north, if the first and last vessel in the sequence are of the same type, then every vessel in the sequence is of the same type.
2. Ships and submarines *do* alternate on distance $N+1$. Thus, for every sequence of $N+1$ vessels sent up north, the first and last vessel in the sequence are of different types.
3. Every vessel sent up north came from south.
4. The controller reacts on sufficient stimulus. Thus, if infinitely many submarines and ships have been sent up from south, then infinitely many submarines and ships are sent up north.

Linear time temporal logic allows to capture these requirements leaving fewer ambiguities. This formalization occupation alone is in fact one of the most rewarding components of a system design. For instance, it may reveal that the above informal requirements leave open whether the controller is allowed to delay vessels of its choice arbitrarily.

However, the focus in this paper is on verification support for checking selected specifications against proposed implementations. We will therefore limit ourselves to the formalization of the two first properties which are invariants. Invariants can be specified with the form $\Box\,\varphi$, where φ is a first-order formula, and $\Box$ is the *always* temporal operator. The judgment $\mathcal{S} \models \Box\,\varphi$ asserts that the formula φ holds on every reachable state of system $\mathcal{S}$.

3 Queues

A closer study of the proposed controller's environment reveal that the queue data-type can be used with advantage to model the flow and temporary storage of vessels.

3.1 Monterey Bay

To properly preserve the wild-life and not disturb the singing (and occasionally doped) amateur pilots in Monterey Bay incoming vessels are stored in the bay in a last-in first-out order. A suitable abstraction of the bay is therefore a *stack*, or equivalently a *list*. Lists have standard constructors ϵ (for the empty list), *cons* to add an element, and selectors *head* and *tail*. With suitable donations from the Packard Foundation, we can always extend the size of the bay and therefore assume it has infinite capacity.

3.2 The *coastal* Mother Road[1]

The flow of vessels is on the other hand more naturally modeled using *channels*. Here the primitive operations consist in appending items to the end of a channel, and taking items out from the front, thus processing

[1] To continue: "66 is the mother road, the road of flight." John Steinbeck, *The Grapes of Wrath*.

elements in a first-in first-out order. In the programming language SPL which serves as one of possible ways to present reactive systems to STeP the statement

$$\mathit{consumer} \Longleftarrow v$$

has as effect to put v in the end of channel *consumer*. The statement

$$\mathit{producer} \Longrightarrow x$$

can be executed when the channel *producer* is non-empty, and has the effect of removing the first element from *producer* and updating x to it.

3.3 Queues as a common denominator

We can kill two birds with one golf-ball (on Pebble Beach) by modeling both lists and channels using queues. Queues are lists with pairs of symmetric constructors and selectors respectively. Thus, symmetric to the constructor `cons` there is a constructor `revcons`, which appends an element to the end of a list instead of to the front as `cons` does. Similarly, symmetric to `head` there is a selector `last`. The functionality of selectors and constructors is illustrated in Figure 2.

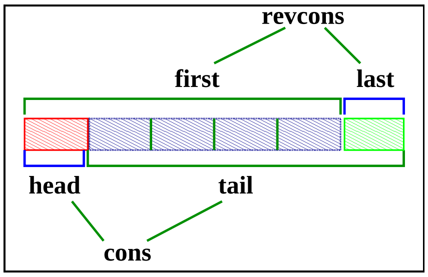


Figure 2: Queue constructors and selectors

3.4 Relations over queues

The most basic relation over queues is undoubtly equality between these. Equality is however in no way the only relation one may want to express between queues. For example, the test whether a queue q contains element e, suggestively written $e \in q$, is another natural candidate. More generally one can assert sub-queue relationships using $q \preceq r$ to assert that queue q is a sub-queue of r. Special cases are when q is a prefix respective suffix of r. These relations are illustrated in Figure 3.

3.5 Decision procedures for queues

Automatic support for queues in quantifier-free formulas with the presented vocabulary has been developed in [2]. A result from this work that we will make use of here is a decision procedure for validity, or equivalently satisfiability, for quantifier free formulas, where terms are built using variables, together with operations

$$\epsilon, \texttt{head}, \texttt{tail}, \texttt{cons}, \texttt{first}, \texttt{last}, \texttt{revcons},$$

and atomic formulas are built using basic relations

$$e \in q,\ \ q = r,\ \ q \preceq r,\ \ \mathsf{prefix}(q, r),\ \ \mathsf{suffix}(q, r)\ \ .$$

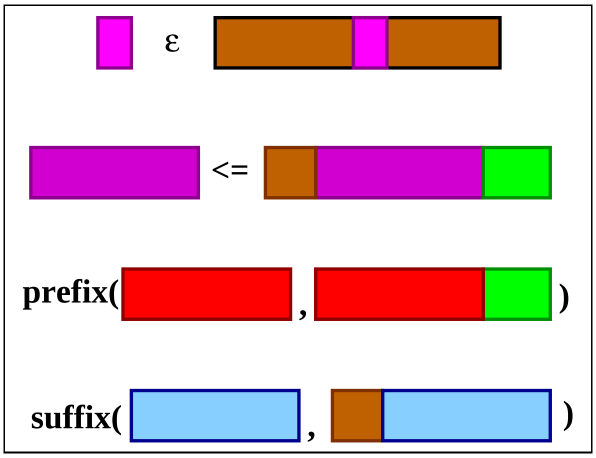


Figure 3: Relations among queues supported by decision procedures

Moreover, the proposed decision procedure can be integrated (tightly) with solvers for other theories, such as arithmetic, to provide a combined validity checker for a union of several theories. Space limitations prevent us from recreating all details here, but readers familiar with interactive verification may appreciate what relief decision procedures give. For example, we can use and reason directly about $\texttt{revcons}(l,a)$ instead of defining it indirectly:

$$\begin{array}{lcl} \texttt{revcons}(l,a) & = & \texttt{rev}(\texttt{cons}(a,\texttt{rev}(l))) \\ \texttt{rev}(l) & = & \texttt{rev2}(l,\epsilon) \\ \texttt{rev2}(\epsilon,l) & = & l \\ \texttt{rev2}(\texttt{cons}(a,l),m) & = & \texttt{rev2}(l,\texttt{cons}(a,m)) \end{array}$$

In this case we would often have to rely on heuristic support for inductive reasoning to unwind the recursive definition of `rev2`.

4 Implementation

Figure 4 suggests an implementation of the controller. It uses a stack to keep track of vessels that cannot be sent north immediately, a counter i to maintain how many vessels of the same value have been sent, and a flag *turn* to record whose turn it is. Symptomatic to the diseases of low-level programming we have used booleans to encode the distinction between sailboats and submarines. So bear in mind that *true* is shorthand for sailboat, and *false* is shorthand for submarine.

$$\begin{array}{lll} \textbf{const} & N & : [1..] \\ \textbf{in} & \mathit{producer} & : \textbf{channel}\,[1..]\ \textbf{of boolean where}\ \mathit{producer} = \epsilon \\ \textbf{out} & \mathit{consumer} & : \textbf{channel}\,[1..]\ \textbf{of boolean where}\ \mathit{consumer} = \epsilon \\ \textbf{local} & \mathit{stack} & : \textbf{boolean list where}\ \mathit{stack} = \epsilon \\ \textbf{local} & \mathit{turn} & : \textbf{boolean where}\ \neg \mathit{turn} \\ \textbf{local} & i & : \textbf{integer where}\ i = 0 \end{array}$$

$$\textbf{procedure}\ \mathit{emit}(v) = \begin{bmatrix} \mathit{consumer} \Leftarrow v; \\ i := \textbf{if}\ \ i+1 \geq N\ \textbf{then}\ \ 0\ \textbf{else}\ i+1; \\ \mathit{turn} := \textbf{if}\ \ i = 0\ \textbf{then}\ \ \neg\mathit{turn}\ \textbf{else}\ \ \mathit{turn} \end{bmatrix}$$

$$\mathit{Producer} :: \begin{bmatrix} \textbf{loop forever do} \\ \quad [\mathit{producer} \Leftarrow \textbf{false or}\ \mathit{producer} \Leftarrow \textbf{true}] \end{bmatrix}$$

$$||$$

$$\mathit{Router} :: \begin{bmatrix} \textbf{local}\ x : \textbf{boolean} \\ \textbf{loop forever do} \\ \quad \begin{bmatrix} \ell_0\text{:}\ \ \textbf{if}\ \mathit{head}(\mathit{stack}) = \mathit{turn}\ \wedge\ \epsilon \neq \mathit{stack} \\ \textbf{then} \\ \quad \begin{bmatrix} \ell_1\text{:}\ \ \mathit{emit}(\mathit{head}(\mathit{stack})) \\ \ell_2\text{:}\ \ \mathit{stack} := \mathit{tail}(\mathit{stack}) \end{bmatrix} \\ \textbf{else} \\ \quad \begin{bmatrix} \ell_3\text{:}\ \ \mathit{producer} \Longrightarrow x; \\ \ell_4\text{:}\ \ \textbf{if}\ x = \mathit{turn} \\ \qquad \textbf{then}\ \ell_5\text{:}\ \ \mathit{emit}(x) \\ \qquad \textbf{else}\ \ \ell_6\text{:}\ \ \mathit{stack} := \mathit{cons}(x,\mathit{stack}) \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 4: CONTROLLER

The implementation is described in a simple programming language with primitives for concurrency and communication via channels. It is one of the possible formats for presenting reactive systems to STeP. STeP compiles the program text into a *transition system*, which models each basic statement of the program as a transition. The transitions (atomic actions) are described using binary relations between the present and next state values of the system's state variables. For example, the statement ℓ_3 is enabled when the control counter ℓ reaches 3 and the channel *producer* is non-empty. It updates the control to $\ell := 4$ and dequeues the first element of *producer*, placing it in the variable x. Other system variables are unchanged. Described as a first-order relation the transition relation reads:

$$\begin{array}{ll} & \ell = 3 \ \wedge \ \mathit{producer} \neq \epsilon \\ \wedge & \ell' = 4 \ \wedge \ \mathit{producer}' = \mathtt{tail}(\mathit{producer}) \ \wedge \ x' = \mathtt{head}(\mathit{producer}) \\ \wedge & \mathit{consumer}' = \mathit{consumer} \ \wedge \ i' = i \ \wedge \ \mathit{stack}' = \mathit{stack} \ . \end{array}$$

Fortunately, STeP provides translations like these automatically and behind the scenes. We use $\mathcal{T}$ to summarize the transitions of system $\mathcal{S}$, τ to access individual transitions, and ρ_τ for the first-order transition relation associated with τ. The initial states of $\mathcal{S}$ are captured using a first-order formula Θ. For our program Θ is

$$\mathit{producer} = \epsilon \ \wedge \ \mathit{consumer} = \epsilon \ \wedge \ \mathit{stack} = \epsilon \ \wedge \ \mathit{turn} = \mathit{false} \ \wedge \ i = 0 \ \wedge \ \ell = 0 \ .$$

5 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 [8]. The most widely used verification rule is (the essentially non-temporal) INV given in Figure 5. It reduces the verification of the invariant $\Box p$ to the first-order verification conditions in premises $B1$ and

$$\begin{array}{ll} & \text{For assertion } p, \\ \text{B1.} & \Theta \rightarrow p \\ \text{B2.} & \{p\}\ \mathcal{T}\ \{p\} \\ \hline & \mathcal{S} \models \Box p \end{array}$$

Figure 5: Basic invariance rule INV.

$B2$. The condition $B2$ is shorthand for

$$\bigwedge_{\tau \in \mathcal{T}} (p(\overline{x}) \ \wedge \ \rho_\tau(\overline{x}, \overline{x}') \ \rightarrow \ p(\overline{x}')) \ .$$

6 Generation of invariants

It is a property of the implementation that the *stack* variable contains only bits of the same value. The alert reader will notice that Figure 1 is misleading in this respect, but keep in mind that we are only analyzing one possible realization of the requirements. We can check the property by postulating the invariant:

$$\varphi_1 : \quad \Box((\neg \mathit{head}(\mathit{stack})) \notin \mathit{stack}) \tag{1}$$

Note that if the *stack* is empty, then the invariant holds trivially although we have left the effect of *head* under specified in this case (not undefined as the programming language centric convention is).

The invariant φ_1 is not inductive, but by using the auxiliary invariants

$$\varphi_2 : \quad at\_\ell_{3,4} \ \wedge \ (\mathit{head}(\mathit{stack}) \leftrightarrow \mathit{turn}) \Rightarrow \mathit{stack} = \epsilon \tag{2}$$

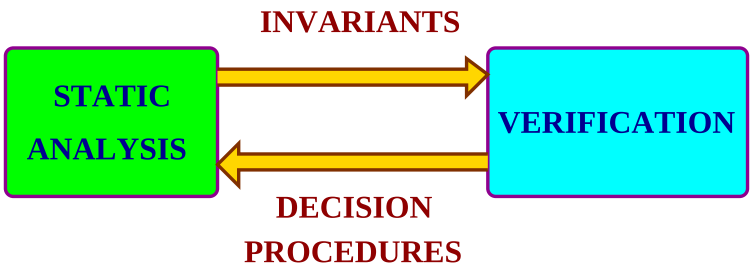
$$\varphi_3 : \quad at\_\ell_{5,6} \ \Rightarrow \ (x \leftrightarrow \mathit{turn}) \wedge ((\mathit{head}(\mathit{stack}) \leftrightarrow \mathit{turn}) \rightarrow \mathit{stack} = \epsilon) \tag{3}$$

we can use rule INV from Figure 5 and the decision procedures for queues to automatically prove the property. We used the shorthand $at\_\ell_{3,4}$ in place of $\ell = 3 \ \vee \ \ell = 4$, and $(\ldots \Rightarrow \ldots)$ as shorthand for $\Box(\ldots \rightarrow \ldots)$.

The invariants φ_2 and φ_3 are generated automatically by STeP and need therefore not be formulated independently by the practicing verifier. Generation of invariants and auxiliary assertions [4] is one of the features STeP provides to bootstrap deductive verification. In this case STeP's user only had to assert the main property to be verified, press a button to generate invariants, and invoke INV to let the decision procedures handle the sub-goals automatically.

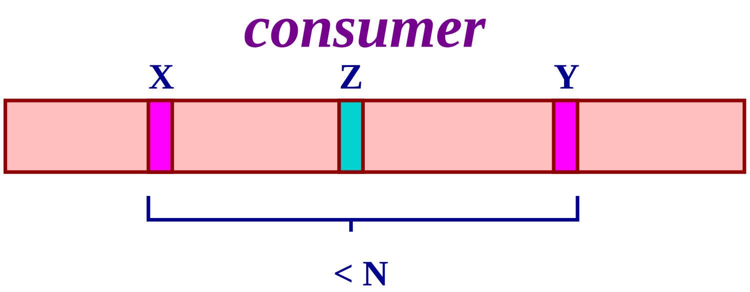
6.1 Abstraction

Invariant generation and decision procedures play an intimate role in an emerging practice in automatic verification of infinite state systems using abstraction [1, 5]. Invariants are generated from abstraction, which relies on techniques from static program analysis. Decision procedures, on the other hand, can be used to generate abstractions preserving more properties as they give more expressive power to the static analyzer. The coupling is loosely motivated by the picture below.



7 Verification

Suppose now that we wish to express that the bits in the consumer do not change value within distance N. Pictorially, if x and y are the same in *consumer*, and the distance between x and y does not exceed N, then any z between x and y must have the same value.



Using a sub-queue relation symbol $\preceq$, operations *head*, and *last*, which pick first and last elements in a queue, and a length measure $|\ |$ we can express this concisely using the invariant:

$$(\forall s)\left(\left(\begin{array}{ll} & s \preceq \mathit{consumer} \\ \wedge & 1 \leq |s| \leq N \\ \wedge & \mathit{head}(s) = \mathit{last}(s)\end{array}\right) \Rightarrow\ (\neg \mathit{head}(s)) \notin s\right) \tag{4}$$

The invariant is unfortunately not inductive, but can be established using the auxiliary invariants below. The predicate *suffix* states that s is a suffix of the queue *consumer*.

$$\Box(0 \leq i < N) \tag{5}$$

$$i > 0 \quad \Rightarrow \quad \mathit{last}(\mathit{consumer}) = \mathit{turn} \tag{6}$$

$$i = 0\ \wedge\ \mathit{consumer} \neq \epsilon \quad \Rightarrow \quad \mathit{last}(\mathit{consumer}) \neq \mathit{turn} \tag{7}$$

$$(\forall s)\left(\begin{array}{l}(\ \mathit{suffix}(s, \mathit{consumer}) \wedge \quad 1 \leq |s| \leq N\ ) \\ \qquad \Rightarrow\ \textbf{if}\ \ |s| \leq i\ \textbf{then}\ \ \neg \mathit{turn} \notin s\ \textbf{else}\ \ (i = 0\ \rightarrow\ \ \mathit{turn} \notin s)\end{array}\right) \tag{8}$$

The good news is on the other hand that verification of both the auxiliary invariants and the main specification proceeds practically automatically thanks to the decision procedures for queues that we develop in the following. The verification condition below is one of the proof-obligations that is produced as a sub-goal by the rule INV and established automatically using the decision procedures.

$$\left(\begin{array}{ll}
 & (0 \leq i \ \wedge \ i < N) \\
\wedge & (0 < i \rightarrow \mathtt{last}(consumer) = turn) \\
\wedge & \left(\begin{array}{l} i = 0 \ \wedge \ \neg(consumer = \epsilon) \rightarrow \\ \quad \neg(\mathtt{last}(consumer) = turn) \end{array}\right) \\
\wedge & \left(\begin{array}{l} \mathsf{suffix}(\mathtt{first}(s), consumer) \\ \ \wedge\ 1 \leq |\mathtt{first}(s)| \ \wedge \ |\mathtt{first}(s)| \leq N \\ \quad \rightarrow \\ \quad\ \mathbf{if}\ |\mathtt{first}(s)| \leq i \\ \quad\quad \mathbf{then}\ (\neg turn) \notin \mathtt{first}(s) \\ \quad\quad \mathbf{else}\ (i = 0 \rightarrow turn \notin \mathtt{first}(s)) \end{array}\right) \\
\wedge & \left(\begin{array}{l} s \preceq consumer \ \wedge \ 1 \leq |s| \ \wedge \ |s| \leq N \rightarrow \\ \quad \mathtt{head}(s) = \mathtt{last}(s) \rightarrow (\neg\mathtt{head}(s)) \notin s \end{array}\right) \\
\wedge & \mathtt{head}(s) = \mathtt{last}(s) \\
\wedge & s \preceq \mathtt{revcons}(consumer, turn) \\
\wedge & 1 \leq |s| \ \wedge \ |s| \leq N \\
\wedge & \left(\begin{array}{l} \mathtt{first}(s) \preceq consumer \ \wedge \\ 1 \leq |\mathtt{first}(s)| \ \wedge \ |\mathtt{first}(s)| \leq N \rightarrow \\ \mathtt{head}(\mathtt{first}(s)) = \mathtt{last}(\mathtt{first}(s)) \rightarrow \\ (\neg\mathtt{head}(\mathtt{first}(s))) \notin \mathtt{first}(s) \end{array}\right)
\end{array}\right) \rightarrow (\neg\mathtt{head}(s)) \notin s$$

Finally, we can verify that elements in distance $N+1$ in the *consumer* are always different using the auxiliary invariant (9) in establishing (10).

$$(\forall s)\,(suffix(s, consumer) \ \wedge i < |s| \leq N \ \Rightarrow\ \mathtt{head}(s) \neq \mathtt{last}(s)) \tag{9}$$

$$(\forall s)\,(s \preceq consumer \ \wedge \ |s| = N+1 \ \Rightarrow\ \mathtt{head}(s) \neq \mathtt{last}(s)) \tag{10}$$

8 Conclusions

So, what was the point? Did we tire the reader with yet another trivial example (albeit not a railroad crossing problem) to discredit formal methods, or at least our version of it? Did we use the *right* way for the requirements capture, as postulated in the abstract? Did we blaze our own trail to to distinguish ourselves, and what can be reused by the community that does not subscribe to temporal logic based deductive verification.

My first purpose has naturally been to describe one of the ways to verify systems with STeP, and in particular highlight decision procedures which is related to my own technical interests. Temporal verification in this flavor consisted of several components: a language for capturing requirements, a language and underlying model for describing systems, and a verification methodology consisting of verification rules, generation of invariants, and decision procedures. The example has been chosen carefully such that standard model checking techniques could not be applied due to the non-fixed parameter N. Deductive methods are largely insensitive to such generalization.

But if one is not interested in verification of temporal requirements of a given system, what do we have to offer? Certainly the decision procedures for queues can handle data-domains independently of whether we are addressing properties of reactive systems, functional programs, or high-level specifications. The decision procedures for queues have even been used in an extensive benchmark for component based software retrieval [6]. Decision procedures offer the algorithmic counterpart of inference from first principles, which impedes the use of any verification technology about anywhere. Decision procedures also often terminate quickly on invalid goals, at least the ones I have had the chance to test. This is particularly helpful in debugging the verification of a specification. Finally, decision procedures, even when implemented only incompletely, is one of the ingredients that can be used for something more fancy than the standard debugging and code optimizing environments for software. If the aim is to program with constraints, decision procedures are again the required core technology.

In summary, I would like to emphasize two points in connection with this paper: First, formalization, which was done only roughly in Section 2, is certainly essential for the documentation and success of the development of a software based system. "See source code for documentation", or "Its all in my mind", are unfortunately not hypothetical scenarios - to the authors own regretful experience. Adapting an independent formalization process allows to communicate and document system requirements and architecture. Formalization should naturally only be carried out to the extent that it contributes to the design and analysis of the constructed system. On the other hand, constructing a system so that it is amenable to formalizable analysis has never been a disadvantage (for me). Second, automatic support of high-level formalizable frameworks adds value to the development of software. This may be in the form of compilers for high-level programming languages (ML, SeqL), constraint-based programming environments, or expressive debugging tools, such as, deductive verification, extended static checkers, and model checkers. The technology for the latter is reaching a stage today where they can be used with some benefit in the design cycle along with standard debugging and development tools. Naturally, decision procedures play a central part of this process.

References

[1] Bensalem, S., Lakhnech, Y., and Owre, S. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [7], pp. 319–331.

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

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

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

[5] Colón, M. A., and Uribe, T. E. Generating finite-state abstractions of reactive systems using decision procedures. In Hu and Vardi [7], pp. 293–304.

[6] Fischer, B. *Automatic software retrieval.* PhD thesis, TU Braunschweig, 1998.

[7] Hu, A. J., and Vardi, M. Y., Eds. *Proc. 10^{th} Intl. Conference on Computer Aided Verification* (June 1998), vol. 1427 of *LNCS*, Springer-Verlag.

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