Technical systems in logic: questions of formalization and automatic verification

A. Nepeivoda

abstract. In the paper technical systems with counters are considered as logical models. The questions of formalization in temporal logic and automatic analysis via computational tree transformations are discussed.

Keywords: temporal logic, verification, formalization, proof constructing

1 Introduction

Some classes of modal logic, such as CTL (computational tree logic) and LTL (linear temporal logic), are now rather widely used in computer science for the task of verifying finite automata [2]. To be verified, an automaton must be rewritten in terms of the corresponding logic, and then some of proof-constructing techniques, automatic or interactive, are applied to find whether the automaton performs actions it is supposed to perform (and no other). If the formalization is sound, this procedure must reveal all the possible errors that can appear in the automaton. The techniques of formalization and analysis of finite automata and automaton programs via temporal logic are widely known as model checking approach. But the sense of temporal logics isn't restricted by only automata or program systems: they can describe any lasting-in-time process, every moment of which can be described by the classical logic. Though some of these processes are very hard to be described (e.g. due to possibility of infinite branching), others have very simple structure, so temporal logics can be implemented to them even easier than to program systems. In particular, it is possible to expand field of applications of the classes of logic to parametrized electromechanical technical

systems. The linear temporal logic was successfully used to perform an analysis of new reagent-dozing mechanism [11]. Despite the fact that results of the analysis were confirmed by an experiment, some problems were discovered in this approach. The hardest one was to find proper proof-constructing instruments.

The matter is that electromechanical systems like the reagent-dozing mechanism are parametrized and their features to be checked essentially depend on these parameters. So in the most general case induction scheme must be included to make the analysis possible (and it was used to perform the analysis in [11]). Moreover, even one-counter automata representation in modal logics can lead to undecidability [3]. But in every specific case we need only very restricted version of the induction. So neither model checking instruments nor interactive provers fit the task perfectly: the former are too weak in some aspects (no induction) and too complex in others (allowing very complicated automaton structures); the latter are too strong and therefore demand human decisions. Our suggestion was to try an universal technique of program transformation on this problem, and the first results of this approach have been received.

The technique is called supercompilation and was developed since seventies of the last century (starting with the works of V.F. Turchin) [15]. Its essence is unfolding a computational tree of a partly specialized program together with folding it back to a program graph. The term "partly specialized" means that some input parameters of a program can be known (and some can remain unknown). The process of unfolding resembles the one that is performed in model checking on automata. In general, supercompilers have no domain-specific transformations to optimize complex finite structures so they cannot compete with corresponding model-checkers but the technique of supercompilation was successfully applied to verify simple parametrized protocols [9, 1] and in some of these applications showed itself even more powerful than the corresponding domain-specific programs. Despite the supercompilation wasn't presented as a proof-transforming technique, it has very deep interconnections with logic, and its applications to the temporal logics CTL and LTL lead to some appealing effects.

Therefore we try to construct a "double permutation" of applying

modal logic to technical systems and computer science to modal logic. The results show that all these domains have very much in common, but still a lot of work remain on this way.

2 Infinite state technical system in LTL

2.1 General features

Suppose that a standard circuitry is enriched by some parametrized elements, such as switching boards, and the system is not only to satisfy some qualitative features but has some quantitative ones. Then it may happen that in states of the system never repeat themselves in time. For example, if we have a reagent-dozing system that spreads a reagent of a volume V_r after a volume $K * V_w$ of water passes through a pipe, then in most moments of time the concentration of the reagent in the water reservoir is unique (it is expressed as a formula $\frac{V_r}{V_w} * \frac{T}{T*K+R}$, where T is the total number of full cycles of the mechanism work and R is the number of water volumes V_w that passed through the pipe after the last reagent spread ($R < K$)). In such cases it is reasonable to examine only changes of system quantitative features but not the features themselves. Because of the discrete nature of the parametrized elements that are included in the scheme, these quantitative requirements can be approximated by rational numbers. Because of finiteness of the scheme the satisfiable requirements of state change must belong to some finite (though maybe very large) set. Since parameters of the parametrized elements can also change, the set of the requirements, though remains finite, becomes even larger.

So the following features of the system are guaranteed:

1. Non-constant finite number of elements;
2. Finite set of different states (without considering quantitative aspects);
3. Finite set of changes of quantitative characteristics of the system on a fixed state.

And the following features are to be checked:

1. Uniqueness of successor state. When human intrusion is forbidden, it is essential to know that the system works without any unpredictable indeterminacy.

2. Existence of successor state. If some reachable state never changes, the system will halt.

3. Repeatability of all significant actions.

4. Return to initial state.

These features make us sure that the system infinitely repeats the loop between two initial states.

The whole description of the system somehow resembles the one of the modular arithmetic. The similarity becomes even greater in practice because of the round construction of switching boards that allows a stepping mechanism to move only forward (and the last switch is followed by the first).

In the logical model of the system we describe two classes of axioms: axioms of stability (forbidding actions to be done without a trigger) and axioms of change (describing possible one-step changes of conditions). For every action possible in the system at least two axioms appear, and one more axiom must be included to describe initial conditions. If the system of axioms is full, the feature of linearity (the uniqueness of successors) can be checked by checking the fact that antecedents in every two axioms of change are disjoint. To prove the features of existence and repeatability it might be useful to find additional invariants of the system. After that the last feature can be checked constructively by building a single iteration starting and ending by the initial state. In every specific case, when all parameters of the scheme are known, it can be done in straightforward way.

2.2 LTL formalization

We will use the full language of LTL with the modalities **X**, **G**, **F**, **U**. Though in the first formalization the modality **U** was used in the model axiomatization, now we try to avoid it to make the model better from the point of view of computer science.

As an example of technical system let us consider a simple dozing mechanism which consists of the single stepping switch, the single pump and the single flow indicator. Every time the flow indicator turns the stepping switch does one step. One of the contacts of the switching network (which can be chosen arbitrarily while mounting the scheme) is connected with the pump. When the stepping switch

reaches the contact (let us call it K) the pump does one move and closes the chain that returns the stepping switch to the first position. This sequence of actions repeats itself potentially forever. It is less complex than the scheme in [11]: now we have only one stepping switch instead of the two — but the analogue of our simplified scheme is also used in practice [16]. So in fact we need to verify some special kind of one–counter (not necessarily determined) automaton in LTL — the problem that is known to be PSPACE–hard in general and that becomes undecidable after adding one more counter [3].

The scheme is parametrized with the two parameters: the dimension of switching network N and the number of the contact K that activates the pump. So we need to use not only single propositional variables but also the array of switch states on the switching network, and not only single axioms but also axiom schemes corresponding to different states of stepping switch. More natural approach is to consider not only boolean variables but also a variable of the type $\mathbb{Z}_N$, and this approach is valid due to the two features of stepping switch construction:

1. The stepping switch is always connected to one contact on the switching network;
2. This contact is unique.

The normal work of the mechanism assumes these two features. But if we want to study disruptions then we must use the model with array of booleans since we assume short circuits to happen on the switching network. We now consider most generic case, and turn to the usage of arithmetic in the program model.

The model variables are the following.

1. R is the state of the indicator switch;
2. C is the state of the switch returning the stepping switch to the initial state;
3. P is the state of pump switch;
4. T is the state of additional switch controlling the pump;
5. W_i is the state of the stepping switch (N is the total number of contacts, so $i \leq N$).

The model axioms are separated in two sets. The first is called axioms of change and describes how a current state can affect its successor. Construction of these axioms can be done straightly from the description of the mechanism work.

1. $R \wedge W_i \Rightarrow \mathbf{X}(\neg R \wedge \neg W_i \wedge W_{i+1})$ (the axiom scheme for all i, $i \geq 1 \wedge i < N$).
2. $R \wedge W_N \Rightarrow \mathbf{X}(\neg R \wedge \neg W_N \wedge W_1)$
3. $\neg R \wedge T \wedge \neg W_K \Rightarrow \mathbf{X}R$. This axiom, when implemented, means that the volume of water V_w have passed through the pipe.
4. $P \Rightarrow \mathbf{X}(\neg P \wedge \neg T)$
5. $\neg P \wedge W_K \wedge T \Rightarrow \mathbf{X}P$. This axiom, when implemented, means that the volume of reagent V_r have been spread into the pipe.
6. $\neg T \wedge W_1 \wedge C \Rightarrow \mathbf{X}(\neg C)$
7. $\neg C \wedge W_i \Rightarrow \mathbf{X}(C \wedge \neg W_i \wedge W_{i+1})$ (the axiom scheme for all i, $i \geq 1 \wedge i < N$)
8. $\neg C \wedge W_N \wedge \neg T \Rightarrow \mathbf{X}(C \wedge T \wedge \neg W_N \wedge W_1)$

The second set of axioms is called axioms of stability and forbids switches to change their conditions arbitrarily. In general it is not obvious how to construct these axioms because they are hidden inside a scheme and not presented explicitly in its description. This fact may lead to mistakes in a scheme (for example, in the mechanism tested in [11] some uncertainty appeared exactly because of the absence of the axiom that forbids the indicator switch to change state during the work of pump).

1. $W_i \Rightarrow (C \wedge \neg R \Rightarrow \mathbf{X}W_i)$ (the axiom scheme for all i, $i \geq 1 \wedge i \leq N$)
2. $\neg W_i \Rightarrow (\neg W_{i-1} \vee C \wedge \neg R \Rightarrow \mathbf{X}(\neg W_i))$ (the axiom scheme for all i, $i > 1 \wedge i < N$)
3. $\neg W_1 \Rightarrow (\neg W_N \vee C \wedge \neg R \Rightarrow \mathbf{X}(\neg W_1))$
4. $T \Rightarrow (\neg P \Rightarrow \mathbf{X}T)$

5. $\neg T \Rightarrow (C \vee \neg W_N \Rightarrow \mathbf{X}(\neg T))$

6. $\neg P \Rightarrow (\neg T \vee \neg W_K \Rightarrow \mathbf{X}(\neg P))$

7. $\neg R \Rightarrow (\neg T \vee W_K \Rightarrow \mathbf{X}(\neg R))$

8. $C \Rightarrow (T \vee W_1 \Rightarrow \mathbf{X}C)$

All these axioms describe the dependence of variables' values in a successor state on their values in the current one. And the last axiom remains to be introduced — the axiom of initial state: $W_1 \wedge \neg R \wedge \neg P \wedge T \wedge C \wedge \forall i(i > 1 \Rightarrow \neg W_i)$.

Now we must formalize the conditions to be checked.

1. Uniqueness of successor state. Only one axiom of change can be implemented at every moment of time.

2. Existence of successor state. At least one of the axioms of change can be implemented at every moment.

3. Repeatability of all significant actions.

   $\forall i(R \wedge W_i \Rightarrow \mathbf{XF}(R \wedge W_i)$

   ...

   $\forall i(\neg C \wedge W_i \Rightarrow \mathbf{XF}(\neg C \wedge W_i)$

4. Return to initial state.

   $W_1 \wedge \neg R \wedge \neg P \wedge T \wedge C \wedge \forall i(i > 1 \Rightarrow \neg W_i) \Rightarrow \mathbf{XF}(W_1 \wedge \neg R \wedge \neg P \wedge T \wedge C \wedge \forall i(i > 1 \Rightarrow \neg W_i))$.

The only task is now to present a fast and efficient method of discovering the features of the formalized system. In [11] it was done by hand though there are some techniques in program transformation that can be (and have been) successfully implemented to perform this work. One of these techniques is the supercompilation which is typically used in optimization and analysis of functional and imperative programs [4, 7], but last years it was also implemented to verification tasks [9, 1]. It was noticed that this technique works well on simple parametrized systems of various natures [8].

3 Proving in LTL via supercompilation

The definition of supercompilation can be found in [14]. Saying informally, it consists of the following techniques:

1. Unfolding a computational tree.
2. Folding some branches of the computational tree by means of generalization.
3. Extracting a residual program from the folded graph.

The first technique unfolds computational tree of a program, by step-by-step driving: implementing its rules on computational states until any functional calls disappear. It is exactly the same technique that is used in model checking while unfolding infinite graph of states of an automaton. This resemblance implied the idea that supercompilation may be used for modal logics.

The second technique belongs to the supercompilation itself and is the heart of the method. Some computational branches can be infinite so it is necessary to stop driving them without reaching a final state. For this need the following technique is developed: if some computational branch is considered to be "dangerous" then driving on it halts. The notion of being "dangerous" can vary in different supercompilers (e.g. it may mean "to be too long" or "to repeat itself"). After discovery of a dangerous state a supercompiler folds the computational branch into a loop using generalization. Generalization unifies a latter computational state of the branch with some former state on it that most resembles the latter state (the notion of "resemblance" can also vary in different supercompilers).

After all computational branches of a program are computed into final states or folded, a residual program is constructed from this folded graph. This step isn't discussed in this paper because it is very dependent from a programming language of a target program.

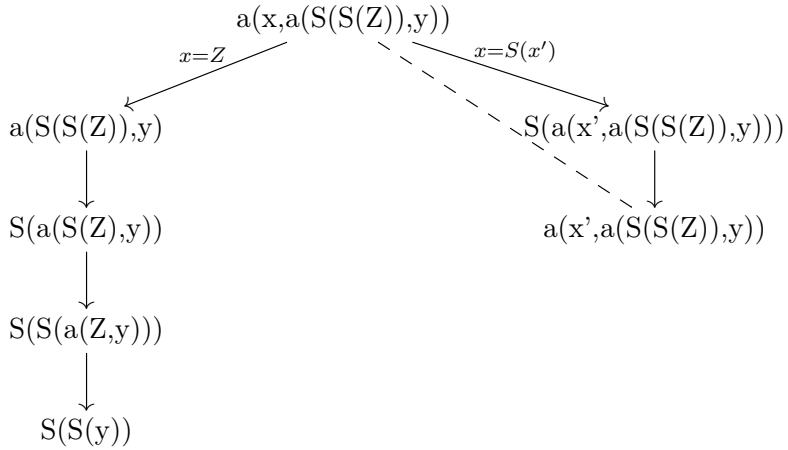
Let us illustrate the process of supercompilation on a simple example from Peano arithmetic.

ПРИМЕР 1.

There is a recursive definition of addition.

```
a(Z,y)=y;
a(S(x),y)=S(a(x,y));
```

Let us supercompile the call `a(x, a(S(S(Z)),y))`.



Every step is an application of a definition or case analysis. The supercompilation lasts until there are no functional calls in a node or the node repeats its predecessor modulo renaming.

Thus, the residual recursive definition corresponding to the call `a(x, a(S(S(Z)),y))` looks as

```
f(Z,y)=S(S(y));
f(S(x),y)=S(f(x,y)).
```

The second addition is already evaluated by the supercompiler so it disappears in the residual program.

In the terms of logic, supercompilation technique rewrites an initial proof for some special case of a theorem (in the example the theorem was an existence of the sum of every two naturals) and then does some obvious logical transformations to receive the proof that isn't more complex than the initial one (it is desirable to make it less complex) but is equivalent to it by means of the set of realizations. There are some restrictions of input proofs: positive supercompilation allows no $\neg$ in disjunctions; perfect supercompilation allows introducing constraints, so it permits $\neg$ in case analysis.

The set of the allowed logical transformations of a proof is very limited and depends on supercompilation technique. For example, the rule $A \wedge A \Leftrightarrow A$ is equivalent to so-called msg (most specific generalization) sharing mechanism and is used in positive supercompilers [13]; the rule $A \vee A \Leftrightarrow A$ is used in the supercompiler

SCP4 [10]. The last uses even more "unsafe" transformations such as $\neg\neg A \Rightarrow A$ that allow to highly optimize very complex programs but in some cases changes their semantics (e.g. transform infinite proofs to finite ones).

ПРИМЕР 2. Consider a recursive function on natural numbers:

```
g(Z)=Z;
g(S(x))=g(h(x,Z));
h(Z,y)=y;
h(S(x),y)=h(x,2^y);
```

`g(x)` is non-terminating for $x > 1$ but in an unsafe supercompiler the observation that all the computations cannot end by the term other than `Z` (so $\forall x(\neg\neg(g(x) = Z))$) may lead to semantics-changing optimization replacing the definition of `g` by a constant zero.

The supercompilation technique doesn't fit well for every task on automatic analysis and has some subtle points such as semantic-changing transformations. But it was successfully used for verification of cash-coherent protocols [9] and communication protocols [1] in cryptography. We decided to try it also on verifying the model of a simple dozer that is described in the previous section.

As input language we chose Refal which is close by semantics to Markov algorithms on strings. It was done because Refal is the input language of the general-purpose supercompiler SCP4 [10]. The convenience of this supercompiler is that it presents perfect information propagation in conditions (allows $\neg$) so it is possible to use not only boolean variables (for which perfect information propagation coincides with positive), but also naturals. SCP4 is not semantics-preserving in case of non-terminating target programs, but if the target program always terminates, this supercompiler preserves its semantics, so the problem is insignificant.

Representation of the model variables repeats the one in the previous section, but we chose unary natural number to represent the state of stepping switch. Representation of the model axioms in programming language was straightforward: we divided data visible by the program on two parts, a current state and a next state. The initial state is completely determined. All the axioms describe the influence of the former to the latter. If after implementing all the axioms some uncertainty remains in the next state, then this uncertainty is represented as free variables which can be replaced

by any value. The axioms of change that correspond to changing the state of system (counting water flow or pumping the reagent) increase corresponding variables: one of them, `e.R`, is the number of volumes V_w that passed through a pipe; another, `e.P`, is the number of volumes V_r that were added to the water flow. The aim is to prove that with any values of free variables the program returns to the initial state and computes the certain pair, `e.R, e.P`, and then to compare $\frac{\texttt{e.P}}{\texttt{e.R}}$ to the technical requirements (it must be equal to $\frac{1}{K-1}$). The supercompilation does this task for any constant N and K. If $K \neq 1$ and $K \leq N$, then the target program is optimized to `'FF'` (that means that $\neg(\texttt{e.R} \neq K-1) \wedge \neg(\texttt{e.P} \neq 1)$). If $K > N$, then no optimization is performed, so the program has indeterminacy and represents the erroneous work of the mechanism. This situation simply means that there is no contact on the switching network that is connected with the pump. The most interesting fact is that for $K = 1$ program isn't optimized to constants, so indeterminacy appears and the system becomes inconsistent. This is an analogue of the error that was found in [11] (and, as was shown experimentally, really can occur in the system). By changing one or another class of axioms we can also see how the system behaves when some physical disruption happens.

But to check all practical cases by SCP4, it is necessary to write simple auxiliary program that lists all of them in the input SCP4 file. This inconvenience appears because modern supercompilers are very careful in transforming programs (or proofs) not to change their semantics (or admit very slight changes) and permit only very few logical transformations. It is supposed that strengthened techniques like distillation [5] can do more complicated transformations like merging several induction proofs in a one so can be able to avoid these restrictions.

4 Conclusion

Our consideration of mechanical models together with logic and computer science led to revealing a deep connection between all these domains. First, mechanical models are easily described via temporal logic with counters, through introducing two classes of axioms that permit and forbid to change condition of an element. Second, the axioms of temporal logic with counters can be easily

rewritten as functions in a programming language strong enough to express negation. Due to the features of quantitative restrictions that can appear in such systems it is possible to represent the conditions of consistency of the scheme as well. And third, it is possible to implement non-domain-specific program transformers to do the automatic analysis of the model. The essence of the last fact is that we must only formulate model axioms and the conditions of correctness and incorrectness of the scheme, and then it is possible to use not proof-constructing, but only proof-transforming technique to verify every special case of the scheme.

This may be the small step to observing the pure theoretical results of reverse mathematics in practice. It is known that folding computational trees only along their branches in supercompilation leads only to linear speedups of residual program [6] (so only $I\Delta_0$-formulas may be eliminated). But embeddings of tree embeddings lead to some non-linear program speedups very close to ones performed by induction [6]. The fact is not surprising from the point of view of logic, but remains very subtle and obscure from the point of view of practical program transformations. Establishing understanding between these two points of view seems to be very fruitful, yet unexplored, domain.

4.1 Acknowledgements

First I would like to thank N.N. Nepejvoda and F.F. Chausov for giving the chance to implement the modal logic to verify a real mechanical system and for the patience that F.F. Chausov showed in the collaboration with a person who is very far from aspects of engineering. Second, the idea of connecting supercompilation with logic first appeared in discussions with Refal community and then was developed in DIKU, Copenhagen. Many thanks to R. Glück for the possibility to do this and for insightful discussions and to A.P. Nemytykh for providing such universal tool as SCP4 which allowed to hold experiments with the model.

Bibliography

[1] *Ahmed, A., Lisitsa, A., Nemytykh, A. P.*: Cryptographic Protocol Verification via Supercompilation. Reachability Problems, 2012. To appear.

[2] *Clarke E.M., Emerson E.A., Sistla A.P.*: Automatic verification of finite-state concurrent systems using temporal logic specifications. In ACM Transactions on Programming Languages and Systems 8(2) (1986) pp. 244–263.

[3] *Demri, S., Lazic, R., Sangnier, A.*: Model Checking Freeze LTL over One-Counter Automata. In Foundations of Software Science and Computational Structures, LNCS Vol. 4962 (2008), pp. 490–504.

[4] *Glück, R., Klimov A.V.*: Occam's razor in metacomputation: the notion of a perfect process tree. In Static Analysis (Proc. Third International Workshop WSA'93), LCNS Vol. 724 (1993), pp. 112–123.

[5] *Hamilton, G. W.*: Extracting the Essence of Distillation. In Proceedings of the 7th PSI, Novosibirsk, Russia, LNCS 5947 (2009): pp. 151–164.

[6] *Hamilton, G.W., Jones, N.D.*: Distillation with labelled transition systems. In PEPM 2012: pp. 15–24.

[7] *Klimov, A.V.*: A Java Supercompiler and its Application to Verification of Cache-Coherence Protocols. In: PSI 2009. LNCS vol. 5947 (2010), pp. 185–192.

[8] *Lisitsa, A., Nemytykh, A. P.*: Verification as a parameterized testing (experiments with the SCP4 supercompiler). In Programming and Computer Software Vol. 33(1) (2007): pp. 14–23.

[9] *Lisitsa, A., Nemytykh, A. P.*: Reachability Analysis in Verification via Supercompilation. Int. J. Found. Comput. Sci. Vol. 19(4) (2008), pp. 953–969.

[10] *Nemytykh, A.P.*: The SCP4 supercompiler: general structure. Moscow, 2007. 152 p. (in Russian)

[11] *Nepeivoda A.*: Verification of a technical system model with linear temporal logic. In Automation and Remote Control, Vol. 93, No. 9 (2012), pp. 124–140.

[12] *Secher, J.P., Sorensen, M.H.*: On Perfect Supercompilation. In Perspectives of System Informatics, LNCS Vol. 1755 (2000), pp. 113–127

[13] *Sørensen, M.H., Glück, R.*: An algorithm of generalization in positive supercompilation. In Logic Programming: Proc. of the 1995 International Symposium (1995), pp. 465–479.

[14] *Sørensen, M.H., Glück, R., Jones, N.D.*: A positive supercompiler. In Journal of Functional Programming, Vol. 6(6) (1996), pp. 811–838.

[15] *Turchin, V.F.*: The concept of supercompiler. In ACM Trans. Prog. Lang. Syst. Vol. 8 (1986), pp. 292-325.

[16] Patent No. 2367819, Russian Federation. MPC F04B 13/00 F17D 3/12 Publ. March 21, 2008. A method and construction for reagent dozing / F.F. Chausov

5 Appendix. Refal representation of the model

```
$ENTRY GO {s.1 s.2 s.3 s.4 (e.1) = <Launch (s.1 s.2 s.3 s.4)(e.1)
(<I 'III'>) (<I 'II'>) (()()('I')'F' 'F' 'T' 'T')>;}
```

`Launch` **checking the consistency condition and applying all axioms*

```
  {(s.1 s.2 s.3 s.4)(e.1)('I' e.N)('I'e.K) (('I'e.KK)('I'e.X)('I')
'F' 'F' 'T' 'T') = <Neq (e.K)'I'e.KK> <Neq ('I'e.X)'I'>;
  (s.1 s.2 s.3 s.4)(e.1)(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P
s.C s.T)=<Launch (s.1 s.2 s.3 s.4)(e.1) <Next <AxiMW <AxiMP
<AxiMR <AxiMC <AxiSC <AxiMT <AxiWStab <AxiTStab <AxiPStab
<AxiRStab <AxiCStab (e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C
s.T) (e.R)(e.P)(e.1) s.1 s.2 s.3 s.4>>>>>>>>>>>>>;
```

```
  }
```

Next *transition to the next moment of time*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT =(e.N)(e.K) ((e.XR)(e.XP)(e.XW) s.XR s.XP
s.XC s.XT);
  }
```

AxiMW *axiom of moving the stepping switch after rotating the flow indicator*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T)(e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.R:'T', <IncR (e.W)e.N>:e.WW = (e.N)(e.K)
((e.R)(e.P)(e.W) s.R s.P s.C s.T) ('I'e.XR)(e.XP)(e.WW) 'F'
s.XP s.XC s.XT;
  e.1 = e.1;}
```

AxiMT *axiom of the pump doing back move*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T)(e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.P:'T'= (e.N)(e.K)((e.R)(e.P)(e.W) s.R
s.P s.C s.T) (e.XR)('I'e.XP)(e.XW) s.XR 'F' s.XC 'F';
  e.1 = e.1;}
```

AxiMP *axiom of the pump doing work move*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T)(e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.P:'F', s.T:'T', <Neq (e.W)e.K>:'F'=
(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T)(e.XR)(e.XP)(e.XW)
s.XR 'T' s.XC s.XT;
  e.1 = e.1;}
```

AxiMR *axiom of making the flow indicator move*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T)(e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.R:'F', s.T:'T', <Neq (e.W)e.K>:'T'=
(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP (e.XW)
'T' s.XP s.XC s.XT;
  e.1 = e.1;}
```

AxiMC *axiom of unlocking the switch C*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T)(e.XR)(e.XP)(e.XW)
```

```
s.XR s.XP s.XC s.XT, s.C:'T', s.T:'F', <Neq (e.W)'I'>:'T'=
(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP 'F' s.XT;
  e.1 = e.1;}
```

AxiSC **axiom of moving the stepping switch due to C*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.C:'F', <Neq (e.W)e.N>:'F' = (e.N)(e.K)
((e.R)(e.P)(e.W)s.R s.P s.C s.T) (e.XR)(e.XP)('I') s.XR s.XP
'T' 'T';
  (e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.C:'F', <IncR (e.W)e.N>:e.WW = (e.N)(e.K)
((e.R)(e.P)(e.W)s.R s.P s.C s.T) (e.XR)(e.XP)(e.WW) s.XR s.XP
'T' s.XT;
  e.1 = e.1;}
```

AxiWStab **axiom of stability of the stepping switch*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.C:'T', s.R:'F' = (e.N)(e.K) ((e.R)(e.P)(e.W)
s.R s.P s.C s.T) (e.XR)(e.XP)(e.W) s.XR s.XP s.XC s.XT;
  e.1 = e.1;}
```

AxiTStab **axiom of stability of the auxiliary switch T*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.T:'T', s.P:'F' = (e.N)(e.K) ((e.R)(e.P)(e.W)
s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW) s.XR s.XP s.XC 'T';
  (e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.T:'F', s.C:'T' = (e.N)(e.K) ((e.R)(e.P)(e.W)
s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW) s.XR s.XP s.XC 'F';
  (e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.T:'F', <Neq (e.W)e.N>:'T' = (e.N)(e.K)
((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW) s.XR
s.XP s.XC 'F';
  e.1 = e.1;}
```

AxiPStab **axiom of stability of the pump switch*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.P:'F', <Neq (e.W)e.K>:'T' = (e.N)(e.K)
```

```
((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW) s.XR
s.P s.XC s.XT;
  (e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.P:'F', s.T:'F' = (e.N)(e.K) ((e.R)(e.P)(e.W)
s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW) s.XR s.P s.XC s.XT;
  e.1 = e.1;}
```

AxiRStab **axiom of stability of the indicator switch*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.R:'F', s.T:'F' = (e.N)(e.K) ((e.R)(e.P)(e.W)
s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW) s.R s.XP s.XC s.XT;
  (e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.R:'F', <Neq (e.W)e.K>:'F' = (e.N)(e.K)
((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW) s.R s.XP
s.XC s.XT;
  e.1 = e.1;}
```

AxiCStab **axiom of stability of the switch C*

```
  {(e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.C:'T', <Neq (e.W)'I'>:'F' = (e.N)(e.K)
((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW) s.XR
s.XP s.C s.XT;
  (e.N)(e.K)((e.R)(e.P)(e.W) s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW)
s.XR s.XP s.XC s.XT, s.C:'T', s.T:'T' = (e.N)(e.K) ((e.R)(e.P)(e.W)
s.R s.P s.C s.T) (e.XR)(e.XP)(e.XW) s.XR s.XP s.C s.XT;
  e.1 = e.1;}
```

I **formatting a string to unary natural number*

```
  {=;
  s.1 e.1 = 'I'<I e.1>;}
```

Neq **checking unequality of two natural numbers*

```
  {() = 'F';
  (s.1 e.0)s.1 e.1 = <Neq (e.0)e.1>;
  (e.1)e.2 = 'T';}
```

IncR **increasing a natural number e.0 modulo e.1, plus 1*

```
  {(e.0) e.1, <Neq (e.0)e.1>:'T' = 'I'e.0;
```

```
(e.0) e.1 = 'I';}
```