AUTOMATED REASONING AND PROVABLY CORRECT DESIGN IN VERIFICATION AND SYNTHESIS

Uni-Kiev Team,
Faculty of Cybernetics, Kiev National Taras Shevchenko University

The Uni-Kiev team plans to continue its investigations, which was carried out under the direction of the RISC in the framework of the Intas 96-0760 "Rewriting techniques and efficient theorem proving". It will continue its research focused on proof-theoretic foundations and on the implementation of tools for finding practically acceptable solutions for the following:

- **The theoretical and practical development of the existent SAD system in the direction of the further construction of intelligent working environment for the automated verification of formal (mathematical and non-mathematical) texts of different kind, mainly, for the verification of a wide variety of properties of software, hardware, and protocols by means of their proving with the help of combining deduction with symbolic computation and model checking.**

- **The development the theory of synthesis and verification of reactive systems (algorithms) with the purpose of the construction various toolkits for designing systems for synthesis of software and hardware, which is based on their declarative specifications and (different) theorem-proving technique.**

In this project, these problems are presupposed to make more coherent than in the previous Intas. It is based on the fact that even in the case of designing of a family of very simple program systems/devices, the verification of some of their properties can often be made only with the help of a general-purpose (universal) automated reasoning system since nobody may forecast what property will be needed to prove (of course, probably except such properties as "safety" and "liveness").

I. THE SAD SYSTEM AS AN AUTOMATED REASONING ASSISTANT

Anatoly Anisimov and Alexander Lyaletski

Faculty of Cybernetics, Kiev National Taras Shevchenko University,
2, Glushkov avenue, building 6, 03022 Kyiv, Ukraine
e-mails: ava@unicyb.kiev.ua, lav@unicyb.kiev.ua

Introduction. Formal methods are widely used in the computer science community. Formal verification and certification is an important component of any formal approach. Such a work can not be done by hand, hence the software that can do a part of it is rather required. The verification methods are often based on a deductive system and "verify" means "prove". Corresponding software is called deductive (proof) assistant.

1. Three Dimensions in Deductive Assistance

To describe the modern approaches, it is convenient to discuss existing mathematical assistants from the point of view of the style of formalization they support, the style of proof they require, and the granularity of proof they accept.

Formalization style. It is determined by the form of definitions (whether they are computable), by the way of reasoning (whether it is constructive, or strictly typed, or calculation-based), by the choice of preliminaries, and so on. Obviously, the style, above all, depends on the choice of base logic and of fundamental theories used for formalization.

At present, there exist two main trends in formalization: applying higher-order logic (type theory) and first-order logic (set theory). Types are used in the majority of the well-known assistant-systems such as Isabelle/HOL, Coq, Omega, PVS, HOL, Automath, Theorema, Lambda-Clam, and others.

Note that the type-theoretic approach favors inductively defined domains and recursive definitions and is well suited to the formalization of concepts of programming or engineering. It may not, however, be the ideal framework for the formalization of traditional mathematics, although most of the systems include a collection of purely mathematical theories.

On the opposite side, the system Mizar uses classical first-order logic and Tarski-Grothendieck set theory. This approach corresponds well to the traditional style of mathematical presentation and the collection of JFM articles

that were verified in Mizar base,constitutes the largest library of mathematical computer knowledge at present.

The SAD system does not adopt any kind of set theory (or any other fundamental theory) as the common base for all formalizations. It prefers to define a particular set of preliminaries for a problem under consideration, choosing base concepts on the appropriate level of abstraction in first-order classical logic.

Proof style. Another important property of a proof assistant is what kind of input it takes. Interactive systems are most often tactic-driven, meaning that a given statement is being proved by a sequence of instructions given to a system. These instructions, tactics, can be primitive, like applying an inference rule, or rather complex, like generating a proof plan for current subgoal or running an external prover. Systems of that type are Isabelle, PVS, Omega, Coq, HOL and others. Working with such a system is easy if it provides a terse set of powerful tactics, which are generally sufficient to capture the desired inferences.

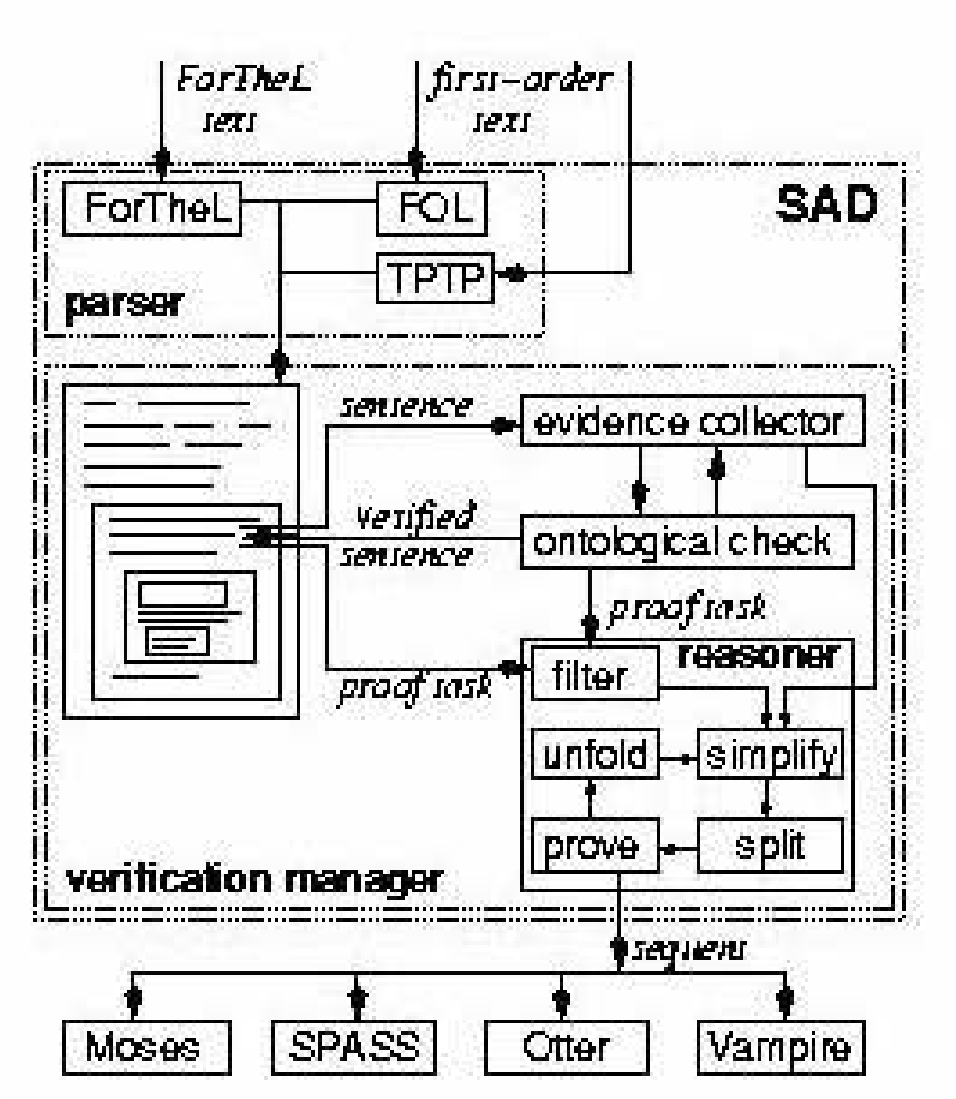
Systems of the second type accept propositions and proofs written in a formal logical language. Of course, this language must be extended with facilities that structure the logical formulas into a proof. The verification system must be capable of checking each successive proof step. Typical representatives of systems of that kind are Mizar and SAD; Isabelle, with introduction of Isar (containing a structured language imitating a language of mathematical proofs) can be considered as a proof-driven system, too.

Proof detailing. The reasoning power of mathematical assistance can be weaker or stronger in accordance with user requirements for proof details. Therefore, systems can range between proof checkers and "proof finders". The former accepts only proof steps having the form of inference rule applications, and, hence, proofs supplied by them have to be fully detailed. Mizar is a system of that kind, though the set of inference rules of Mizar is quite large.

Systems, which we call "proof finders", use proof search methods or proof planning technique and try to close the "gaps" in proofs. The systems SAD, Theorema, Nqthm, and ACL2 are systems of that kind.

Tactic-driven systems usually have enhanceable sets of tactics, so that their "reasoning power" is not peculiar. Almost any tactic-driven system can be classified as a proof finder. However, some experiments with Isabelle and Coq show that whenever one tries to prove a complex theorem ``at once'', without splitting it to a number of ad-hoc lemmas, without using special tactics and existing libraries, the ``proof construction dialog'' quickly becomes complex, wide-branched, and hardly traceable.

2. The SAD System: What can it do?



The SAD Architecture

Having the SAD system located in the "three-dimensional space", we provide a short description of its current state. The SAD can:

- **perform a sequent inference search in classical first-order logic;**
- **prove theorems in the environment of self-contained mathematical texts written in a formal language called ForTheL. This language is a formal proof-writing language, but it is very close to usual mathematical language;**
- **Verify self-contained mathematical ForThel-texts containing propositions and their proofs.**

Generally speaking, SAD verifies/proves the validity of a given input ForTheL text/theorem in an environment which, like any usual mathematical text, consists of definitions, assumptions, affirmations, theorems, proofs, etc.

What does ``correctness'' of such an object mean? We distinguish three types of correctness of a ForTheL text: syntactical, ontological and logical.

Syntactical correctness (well-formalness) is checked by the parser and is a necessary condition for any further actions.

Ontological correctness means that the text in question contains no occurrence of a symbol (constant, function, notion or relation) that comes from nowhere. Every such symbol must be either a signature symbol or introduced by a correct instance of a definition.

Logical correctness is imposed on particular affirmations in the text: theorems, lemmas, and intermediate statements in proofs. Any such affirmation must be deducible from its logical predecessors.

Ontological and logical correctness are, to some extent, independent. It is quite obvious that an ontologically correct text may contain false affirmations. Also, an ontologically incorrect text may appear to be logically correct: e.g., we may not know anything about the relation P and the constant c, yet prove (P(c) --> P(c)).

Nevertheless, it is preferable to require an input text to be ontologically correct for the following reasons. First, ontological verification helps to indicate flaws in your formalization (similarly to type checking in programming languages). Second, during ontological verification the system obtains some important knowledge about the text, which will be used later in logical verification.

3. The SAD system: How can it do that?

Look at the given Figure. All the principal components of the SAD system are shown there.

PARSER accepts a ForTheL text, checks its syntactical correctness and converts the text into a normalized form that will be convenient for further processing.

Verification manager makes her round through the normalized text sentence by sentence. Every sentence is first sent to the ``evidence collector'' which accumulates so-called term properties for the term occurrences in the sentence.

Term properties are literals that tell us something important about a given term occurrence. A literal (i.e. an atomic formula or its negation) L is considered to be a property of a term t in a context Γ (usually, a set of logical predecessors of a given occurrence of t in the text), whenever t is a subterm of L and L is deducible in Γ.

The most important purpose of term properties is to hold information about term ``types'', which is usually expressed by an atomic statement of the form ``$t is a <notion>''.

Fortified with the found properties, the sentence is passed through the ontological checker. Then, if the sentence is an affirmation to be proved, the manager forms a kind of sequent (we call it proof task) and sends it to the reasoner. Note that the ontological checker may also resort to the reasoner in order to find whether the guards of a given definition or signature extension are satisfied.

REASONER can be viewed as a kind of automated heuristic based prover, supplied with a collection of proof task transformation rules. This collection is not intended to form a complete logic calculus. The purpose of the reasoner is not to find the entire proof on its own, but rather to simplify inference search for the background prover. The latter is a combinatorial automated prover in classical first-order logic, whose duty is to complete the proofs started by the reasoner. If the background prover fails to find the inference at some instant, the reasoner may continue the proof task transformation or try an alternative way, or just reject the text.

MOSES is a background prover. It bases on the original sequent formalism satisfying the following requirements:

- **the syntactical form of an initial problem should be preserved;**
- **inference search should be goal-oriented;**
- **deduction must efficiently be made in the signature of an initial theory;**
- **equality handling should be separated from the deduction process.**

These features of the sequent formalism become very important when some interaction between a user and the reasoner and/or the prover is presupposed, as well as when the help of external services really is needed for tasks that require the "sophisticated" technique of their solving. Besides, it permits to incorporate natural reasoning methods. Additionally, it gives a possibility for constructing efficient computer-oriented methods of logical inference search in intuitionistic logic.

Note that an external theorem proving system like Otter, SPASS, or Vampire may be used. This capability of SAD provides us with a (yet another) scale to compare automated theorem provers: trying them on relatively simple problems in complex and heavily redundant contexts rather than on hard problems with a pre-adjusted set of relevant premises (mostly the case for problems in the famous TPTP library).

4. Experiments

In the course of development of SAD, we have conducted a number of essays on formalization and verification of non-trivial mathematical results:

- **Ramsey's Finite and Infinite theorems;**
- **The stability of a refinement relation over a number of operations on program specifications;**
- **Some properties of finite groups;**
- **The Cauchy-Bouniakowsky-Schwarz inequality;**
- **The square root of a prime number is irrational.**
- **The Chinese remainder theorem and Bezout's identity in terms**
- **of abstract rings;**
- **Tarski's fixed point theorem (cited above).**

The texts listed above were written in ForTheL and automatically verified in SAD (using SPASS as the background prover). This work have taught us many important lessons. Let us mention some of them:

- **A formalization style is critical: the choice of symbols to introduce in definitions, the choice of preliminary facts, and even the way a proof is structured may decide whether the text will be verified or not.**
- **It is highly desirable to comprehend the proofs before writing them in ForTheL. The SAD system may succeed to fulfil the gaps in a well thought-out reasoning, but it will not invent one for you.**
- **In most cases, the background prover finds the proof in three seconds --- or does not find it at all.**

5. Example of ForTheL-text

Definition DefCLat. A complete lattice
is a set S such that every subset of S
has an infimum in S and a supremum in S.

Definition DefIso.
f is isotone iff for all x,y << Dom f
x <= y => f(x) <= f(y).

Theorem Tarski.
Let U be a complete lattice.
Let f be an isotone function on U.
Let S be the set of fixed points of f.
S is a complete lattice.
Proof.
Let T be a subset of S.

Let us show that T has a supremum in S.
Take Q = { x << U | f(x) <= x and
x is an upper bound of T in U }.

Take an infimum q of Q in U.
f(q) is a lower bound of Q in U.
f(q) is an upper bound of T in U.
q is a fixed point of f.
Thus q is a supremum of T in S.
end.

Let us show that T has a supremum in S.
SAD does not support proofs by analogy,
so we have to repeat here our reasoning.
...
end.
qed.

Note that the corresponding ForTheL-text contains 11 statements in preliminaries (ordered sets), 7 definitions (upper and lower bounds, supremum, infimum, complete lattice, isotone function, fixed point), 2 lemmas, 18 sentences in the proof of the theorem.

6. Some Remarks on SAD

Certainly, we could not give here a detailed description of all nice features of SAD. SAD is a powerful system and its power lies in its reasoning facility. Experiments show that, for example, the specific strategy of definition processing contributes a lot to the success of the whole verification process. If we use definitions straightforwardly --- convert them into formula images and add the corresponding premises to the sequent that goes into a prover --- we have no chance to verify the proof of Tarski fixed-point theorem as it is formulated above, even when the winner of CASC competitions is chosen as the background prover.

SAD is not a perfect system (as any another!). One can easily see how it may be improved and developed.

Our nearest research and implementation plans are:

- **To extend ForTheL and SAD with some means to talk and reason about second-order objects (functions, vectors, sequences) and operations on them;**
- **To provide users with a facility to implement custom strategies for the reasoner;**
- **To improve the existent possibilities of the background prover in the direction of using efficient handling of equality and of applying different non-classical logics.**
- **To develop and implement a mathematical library of SAD to accumulate verified portions of mathematical knowledge and to support further (deeper) advances in formalization.**

7. Where Can the SAD System Be Useful?

In any domain where precise deductive style formalism is appreciated as the means of problem description. (Note that the problem formalization is always the hardest part of the whole work. Right formalization is a 80% guarantee of a successful verification.) In particular, these domains are:

- **automated theorem proving,**
- **the verification of mathematical papers,**
- **online training in mathematics and logic,**
- **construction of knowledge bases for formal theories,**
- **integration of symbolic calculation with deduction.**

Also, it can be adapted

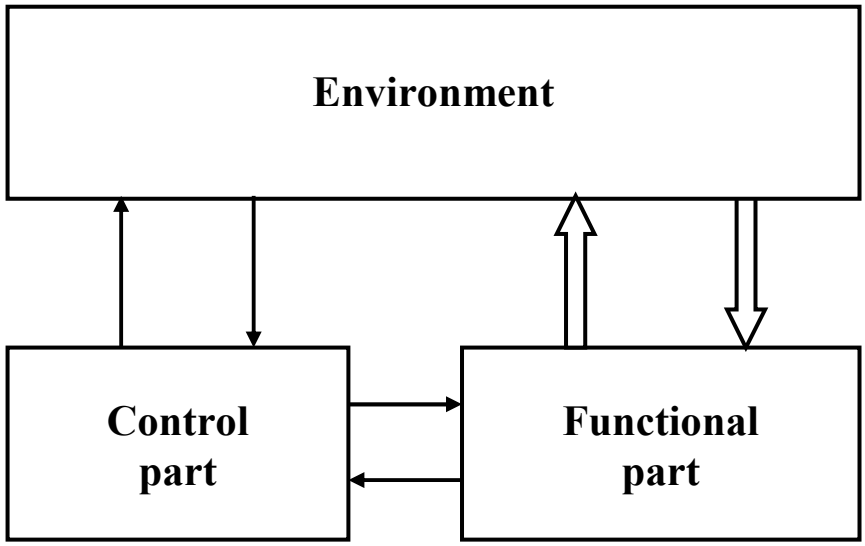
- **for solving logical problems of decision-making theory,**
- **for verifying formal specifications of both software and hardware, and so on.**

In the long run, the ideas being used in the SAD system and its tools aim to construct and support a powerful computer- and knowledge-based (Internet) infrastructure for mathematical research and education, as well as, for deductive processing of formalized text of appropriate kind.

II. SYNTHESIS AND VERIFICATION OF REACTIVE ALGORITHMS

Glushkov Institute of Cybernetics of NASU,
40, Glushkov avenue, 03680 Kyiv, Ukraine
e-mail: ancheb@gmail.com

We are going to focus our attention on the problems of the synthesis and verification of reactive systems (algorithms).



Reactive System

By reactive systems (algorithms) we mean systems (algorithms) that perform step-wise transformations of infinite input sequences into infinite output sequences. Algorithms of that sort describe the behavior of systems continuously interacting with their environment. Formal methods based on logical specification and automated theorem proving are very promising for the development of such systems.

1. Basic Approaches

It is possible to select two basic approaches to the development of correct reactive algorithms: the formal verification of the informally obtained high-level implementation of an algorithm; and the provable synthesis of an algorithm (software/hardware system), which uses of a rigorous automated reasoning technique that convert a declarative specification of requirements to an algorithm into a high-level imperative representation of the algorithm.

Formal verification proves that the (informally obtained) imperative representation of an algorithm has certain specified properties, but does not yet guarantees that the selected representation will exhibit the intended behavior.

Synthesis uses rigorous methods to transform the declarative specification of requirements to an algorithm into its imperative representation and guarantees the precise correspondence between the specification and its imperative implementation.

2. Provably correct design

We address the provably-correct design of synchronous reactive algorithms from their declarative specification. All design transformations are proved to be correct and they are performed automatically. Every change in the algorithm representation made by a designer during interactive design process is verified for correctness (with respect to the initial specification).

REMARK. There are two main classes of these properties: safety properties and liveness properties. A verification-oriented language should have facilities for expressing both safety and liveness properties, while in a synthesis-oriented language we can make do with the facilities for expressing only safety properties.

In this project, we will focus our attention on the provably correct design of software and hardware. For this purpose we will use a special initial specification language being a subset of the first-order language with monadic predicates interpreted over the set of integers which is regarded as a discrete time domain.

Semantics of a reactive system specification, as it follows from the above-given figure, can be described in the form of two interacting non-deterministic automata. One of them specifies the behavior of the system under development and the other specifies the behavior of the system's environment, or more precisely, available information about the behavior of this environment.

3. New contributions to the project

The previous Intas 96-0760 followed the above-mentioned approach and the new Intas project will make the same.

The new contributions **may be classified as follows:**

- **Construction of tools for checking specification for consistency.**
- **Development of new methods for synthesis of reactive algorithms from logical specifications.**
- **Inductive synthesis (according to the structure of the specification formula).**
- **Synthesis from the formula represented by the set of clauses.**
- **Synthesis by constraints propagation.**
- **Verification of liveness properties of synthesized models.**
- **Development of synthesis techniques for Buchi automata with minimal number of states**
- **Simplification of Buchi automata (state minimization).**
- **Development of reduction technique for the automaton modeling the system to be verified.**