Proceedings of the

Thirteenth Workshop on Automated Reasoning

3rd – 4th April 2006

The University of Bristol, United Kingdom

Contents

LCF-style Verification with a SAT-solver

Hasan Amjad*
*University of Cambridge Computer Laboratory
15 J J Thomson Avenue, Cambridge CB3 0FD, UK
Hasan.Amjad@cl.cam.ac.uk

Abstract

We present an integration of the HOL theorem prover with the MiniSat SAT-solver. In addition to earlier work, the interface allows efficient reconstruction of proofs of unsatisfiability within HOL, using MiniSat proof logs.

Introduction

The recent past has seen increasing adoption of formal methods for hardware verification by the industry, for verifying microprocessor components, System-on-Chip protocols and embedded control software. This has brought increasing pressure on the formal methods research community to continually develop formal techniques that can scale to industrial problems.

In such a state of continuous development of an automated verification tool, with competitive pressure to improve time-to-market, the power and speed of the tool may be focused on at the expense of maintainability or even correctness. Yet high-assurance verification frameworks require time to develop and often sacrifice performance for assurance.

Recent years have seen a trend towards tool combination, where specialised high-performance verification engines are integrated within more mature frameworks. The hope is to combine the maturity of the framework with the performance of the specialised tool.

One example of a mature framework is the HOL theorem prover (4). HOL is a mechanization of higher-order logic (Church's simple theory of types plus Hindley-Milner polymorphism) and has been used in many significant verifications. It is "LCF-style", i.e., only a very small core of code is soundness-critical. Users can extend the non-core framework without affecting soundness. This high assurance comes at a price: all new code must ultimately delegate to the core. Shortcuts are not allowed, severely restricting the scope for optimization.

A SAT-solver is a specialised high-performance tool for deciding propositional satisfiability (the SAT problem). The general problem is decidable but NP-complete. Many important problems in verification can be solved by representing them as instances of the satisfiability problem and invoking a SAT-solver. MiniSat (2) is a well known SAT-solver that is able to produce a proof in case of unsatisfiability.

We assume the reader is familiar with the resolution principle in automated reasoning, and its attendant terminology.

Our Contribution

For satisfiability checking, integrating MiniSat into HOL is simple. If a problem is satisfiable, we can confirm that the satisfying assignment supplied by the SAT-solver is indeed a model by substituting into the propositional formula and evaluating to truth.

However, many SAT-based algorithms rely rather on the problem being unsatisfiable. They can then conclude, by decidability of SAT, that the negation of the problem is a tautology. Here, a SAT-solver's failure to find a model is considered proof that the problem is unsatisfiable.

Unfortunately, in HOL we do not have the luxury of trusting the SAT-solver since it is not part of the soundness-critical core. Recently, MiniSat has been equipped with proof-logging, whereby a proof of unsatisfiability can be obtained from the solver. The input term to MiniSat must be in conjunctive normal form. The proof is then output as a ground resolution refutation over the clauses of the input term.

We show that this proof can be replayed within HOL in an efficient manner. We use four optimizations to speed up the reconstruction:

1. The MiniSat log is essentially an activity log, and we first filter out single-step resolutions in the log that are not actually used in the proof. This is easily done by a sink (i.e., the empty clause) to sources (i.e., the problem clauses) traversal of the resolution graph.

2. Clauses are represented in a "dualised" form within HOL, intuitively as two-sided sequents with empty right-hand sides. This obviates tedious associative-commutative reasoning when manipulating clauses. Thus for instance the clause represented by the HOL term $x \vee \neg y$ can be dualised as the HOL *theorem* $\{\neg x, y, x \vee \neg y\} \vdash F$ where F

denotes falsehood. A single resolution step can then be replaced by a single application of the cut rule (cutting out the literal that is resolved upon).

3. Chains (i.e., non-branching sub-trees) within the resolution graph can be shortened, using dynamic programming. Clauses within the chain are reordered to minimise the size of intermediate resolvents. The only restriction observed is that cut literals not be reintroduced. This also addresses an inefficient corner case of MiniSat proof logging, where the same litereal is resolved on more than once within a single chain.

4. Some redundant resolutions can be eliminated by considering the cut-rule occurances of the resolution graph represented as a sequent proof. The basic idea is similar in spirit to the "connections" methods of Kowalksi, Bibel, Andrews and others (1). However, since the aim is proof compression rather than proof search, the usual heuristics (e.g., unit clause preference) are not much help and often make the situation worse. The algorithm proceeds by first eliminating those cut rule occurances that create maximum simplification of the resolution graph. The selection criterion are greedy, e.g., maximise the reduction in the size of the resolvent.

Earlier work in this vein (3; 5) also used steps 1 and 2 above. To the best of our knowledge, steps 3 and 4 represent original work.

Conclusions

Our prototype implementation dramatically outperforms the existing propositional tautology checkers in HOL. For a mixed problem set drawn from the TPTP and older DIMACS challenge problem sets, the existing checkers timed out on most of the bigger problems where our checker was able to present the proof within a minute. On smaller problems where the existing checkers terminated within time, our checker was ten to a thousand times faster.

Many hardware verification problems can be represented as satisifiability problems. Indeed, as an experiment, we have checked an arbitration property of a HOL model of the AMBA AHB[1] System-on-Chip protocol using the MiniSat interface. We hope to extend this work to the rest of the model in due course.

References

[1] W. Bibel. On matrices with connections. *JACM*, 28(4):633–645, 1981.

[2] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[3] Pacal Fontaine, Kamal Gupta, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Towards a combination of heterogeneous deductive tools for system verification. APPSEM II Workshop, September 2005. Frauenchiemsee, Germany.

[4] HOL. The HOL-4 Proof Tool. Tool URL: `http://hol.sf.net`, 2003.

[5] Tjark Weber. Using a SAT solver as a fast decision procedure for propositional logic in an LCF-style theorem prover. Technical Report PRG-RR-05-02, Oxford University Computing Laboratory, August 2005. Theorem Proving in Higher Order Logics - Emerging Trends Proceedings.

2

[1] ©ARM Limited 1999.

Specification and Verification of Reconfiguration Protocols in Grid Component Systems

Alessandro Basso*, Alexander Bolotov, Artie Basukoski, Vladimir Getov

*Harrow School of Computer Science
University of Westminster
United Kingdom
A.Basso@student.westminster.ac.uk

Ludovic Henrio†
†CNRS/I3S/INRIA
Sophia Antipolis
France
ludovic.henrio@sophia.inria.fr

Mariusz Urbanski‡
‡Institute of Psychology
Adam Mickiewicz University
Poznan
Poland
murbanski@amu.edu.pl

Introduction

There are two approaches to building long-lived and flexible Grid systems: exhaustive and generic. The former approach provides rich systems satisfying every service request from applications but consequently its implementation suffers from very high complexity. In the latter approach, we represent only the basic set of services (minimal and essential) and thus overcome the complexity of the exhaustive approach. However, to achieve the full functionality of the system, we must make this lightweight core platform reconfigurable and expandable. One of the possible solutions here is to identify and describe the basic set of features of the component model and to consider any other functions as pluggable components which can be brought on-line whenever necessary (9).

Establishing the theoretical foundations of the generic processes involved in designing and functioning of such Grid systems is highly important. Significant part of this research lies in the area of formal specification and verification of the core component model and the properties of the desired Grid systems.

Fractal Component Model

We depart from the Fractal component model (6), a modular and extensible component model. The Fractal specification defines a set of notions necessary for the model, an API (Application Program Interface), and an ADL (Architecture Description Language).

Components are characterized by their *content* and the *membrane*. If the content of a component is hidden (in which case it is simply a black box) we call it *primitive*. If a component is constituted by a system of some other components (sub-components) it represents a *composite* component controlled by the membrane.

Fractal is a multi-level specifications. Depending on their conformance level, Fractal components can feature introspection and/or configuration. The *control* interfaces are used in the Fractal model to allow configuration (reconfiguration), and are defined as *non functional*. A *functional* interface can provide the required functionalities and we call it the *server* interface. Alternatively, a *client* interface requires some other functionalities. Component interfaces are linked together by *bindings*.

Reconfiguration is obtained by triggering appropriate actions on three interfaces: life-cycle, binding, and content control interface. A reconfiguration can be triggered by any component that has a reference to a correct non-functional interface.

In this work we focus on predefined categories of reconfigurations and on proving properties on these reconfiguration. As far as the reconfiguration is concerned we use the classical assumption that replacing a component by a similar one is safe for the system.

Specification of the Fractal component model in temporal logic

We provide the formal framework for the components in a specific branching-time temporal logic, or $\mathrm{SNF}_{\mathrm{CTL}}$(Separated Normal Form for Computation Tree Logic) (5), and then directly apply temporal resolution as a deductive verification tool.

Taking into account potential applications of CTL-type logics in specification and verification of concurrent and distributed systems, two combinations of future time temporal operators, $\Diamond$ ('sometime') and $\Box$ ('always'), are useful in expressing *fairness* (7): $\Diamond\Box p$ (p is true along the path of the computation except possibly some finite initial interval of it) and $\square\diamond p$ (p is true along the computation path at infinitely many moments of time). It has been shown that $\mathrm{SNF_{CTL}}$ can express these simple fairness constraints and their Boolean combinations (2; 3). In (1; 2) a clausal resolution over the set of clauses $\mathrm{SNF_{CTL}}$ has been defined while the search strategies for this method were presented in (4).

These developments allow us to argue on behalf of the following formal framework to reason about the configurations/reconfigurations protocols of a Grid component model:

$$FCM \longrightarrow ECTL^{+} \longrightarrow SNF_{ctl}(FCM) \longrightarrow BTR$$

Here we suggest the specification of the Fractal component model (FCM) in the logic $ECTL^{+}$ with the subsequent translation into $\mathrm{SNF_{CTL}}$(FCM), the $\mathrm{SNF_{CTL}}$ based formal specification of FCM, and application of the 'branching temporal resolution' method (BTR), the temporal resolution technique defined over the set of $\mathrm{SNF_{CTL}}$ clauses.

Example Specification

Let us consider a simple printing queue component model which consists of a client and one printing queue component as primitives. The client interfaces of the client are of type CI_a and the server interfaces of the printing queue are of type SI_r. In the fractal model four controller interfaces are defined, for reasons of space, we will only specify the safe-unbinding part of a reduced Life-Cycle Controller (LCC) so that it can be used in the deductive reasoning (Figure 1).

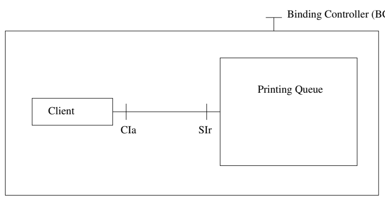

Figure 1: Example in Fractal

Dealing with non-functional interface. Let the propositions $Prim_1, \ldots, Prim_n$ refer to all our primitive controllers. Each $Prim_i$ has a form $Prim_n(CI_a, SI_r)$, i.e. when true it specifies that a Primitive with Client Interface CI_a is bound to the Server Interface SI_r. In our example we have two primitives, one for the Printing Queue and one for the Client using the Printing Queue.

LCC is a proposition which when true signifies that the Life Cycle Controller is active.

Before introducing the Life Cycle Controller Formula we would need to specify how components are started and stopped. Here we will only provide a partial specification of the Life Cycle Controller and two primitive components; we only deal with the formula that captures the bindings of the two components. We will model the start of the components by attaching them to **start**.

Now we introduce the specification of the Simplified Life-Cycle Controller:

$$\neg LCC \wedge \neg(Prim_1(CI_a, SI_r) \vee Prim_2(CI_a, SI_r)) \Rightarrow$$
$$\mathbf{A}\square LCC \Rightarrow (Prim_1(CI_a, SI_r) \wedge Prim_2(CI_a, SI_r))$$

which states that if neither of the components are bound and the LCC is not active then in all possible computations when the LCC is active then we must have the two components bound.

Specification of reconfiguration scenarios.

When a controller needs to dynamically reconfigure the scenario, some properties that we assumed as always true may not hold anymore, for example a component might be removed from the model, or another added, making the system to have to take into consideration other possible paths which have just opened.

The Client can set a request for printing of the type $req(CI_a, SI_r)$. When true, this proposition states that a printing request has been raised by the client which has binding (CI_a, SI_r). The Printing queue specification is represented as $print(CI_a, SI_r)$ which states that a printing request has been satisfied by the printer which has binding (CI_a, SI_r). We will take in consideration the safety part of the specification and its requirements (8). Using the specifications of the components we are able to generate a complete specification for the example considered. To verify the properties of this specification we extract relevant sets of $\mathrm{SNF_{CTL}}$ clauses (3) and then apply a clausal temporal resolution method (5).

References

[1] A. Bolotov. Clausal resolution for extended computation tree logic ECTL. In *Proceedings of the Time-2003/International Conference on Temporal Logic 2003*, Cairns, July 2003. IEEE.

[2] A. Bolotov and A. Basukoski. Clausal resolution for extended computation tree logic ECTL. *Journal of Applied Logic*, in Press.

[3] A. Bolotov and A. Basukoski. A Clausal Resolution Method for Branching Time Logic ECTL+. To be published in Annals of Mathematics and Artificial Intelligence, Springer Verlag.

[4] A. Bolotov and A. Basukoski. Search Strategies for Resolution in CTL-Type Logics: Extension and Complexity. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning, (TIME 2005)* 195 - 197, IEEE Computer Society, 2005.

[5] A. Bolotov and M. Fisher. A Clausal Resolution Method for CTL Branching Time Temporal Logic. *Journal of Experimental and Theoretical Artificial Intelligence.*, 11:77–93, 1999.

[6] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proc. of the 7th Int. Workshop on Component-Oriented Programming (WCOP02)*, 2002.

[7] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics.*, pages 996–1072. Elsevier, 1990.

[8] Z. Manna and A. Pnueli. Temporal Specification and Verification of Reactive Modules. Weizmann Institute of Science Technical Report, March 1992.

[9] J. Thiyagalingam, S. Isaiadis and V. Getov. Towards Building a Generic Services Platform: A Components- Oriented Approach. In: V. Getov and T. Kielmann, eds. Component Models and Systems for Grid Applications, Springer-Verlag, 2004.

A Framework for Describing Information Providing Web Services

Andrey Bovykin⋆ Evgeny Zolin†
⋆Department of Computer Science, University of Liverpool, UK, andrey@csc.liv.ac.uk
†School of Computer Science, University of Manchester, UK, ezolin@cs.man.ac.uk

Abstract

We introduce a formal framework for describing stateless *information providing* Semantic Web Services using Description Logic. The extended notion of service description includes, besides the types of its inputs and outputs, a specification of relationships between inputs and outputs, which has the form of a conjunctive query. From syntactic point of view, it is an extension of the way services are described in OWL-S Service Profile. However, semantically, the definition of *service matching* we introduce here yields a high precision service discovery algorithm. We show that the reasoning problem of service matching for this kind of descriptions is reducible to checking subsumption between two conjunctive queries w.r.t. an ontology, which is a standard reasoning task.

Introduction

The aim of this work is to provide a formal framework for describing stateless information providing web services. When such a service is executed, it accepts from a user an input data of a specified format ("typed data") and returns back to the user some information as an output. Most services of this kind are *stateless*, i.e., they only provide information about the current state of the world, but do not change that state.

We develop our framework in the context of *service discovery* problem: given a description of a service Q that is to be found (called a *service request*), a search engine tries to match the request to service descriptions stored in a repository (they are called *service advertisements*), using its reasoning system and based on the background *ontologies* to which the request and the advertisements refer to. Therefore, our task is to formulate the reasonable conditions of *service matching* that allows for high precision service discovery and such that the matching problem can be decided automatically.

Let us point out the distinguishing features of this framework. Firstly, a description of a service is entirely based on standard background ontologies (i.e., we do not introduce any concepts or roles specially dedicated to describing web services). Therefore we can reuse the existing end emerging ontologies, and, most important, we can use the semantics defined by these ontologies for service matching purposes. Secondly, since the services we describe are stateless, their descriptions need not to contain pre- or post-conditions. Thirdly, the notion of service matching formalised in our framework is decidable and reducible to a standard reasoning task, so one can reuse the existing reasoning systems that are capable to solve the corresponding reasoning problem.

Describing services

Definition 2.1 (Service description). A *service description* is an expression of the form $S := \langle \vec{x}:\vec{X};\ \ \vec{y}:\vec{Y};\ \ \Phi(\vec{x},\vec{y})\rangle$, where $\vec{x}:\vec{X}$ and $\vec{y}:\vec{Y}$ are tuples of *input* and *output* variables together with their types (which are DL concepts), and $\Phi(\vec{x},\vec{y})$ is a *conjunctive query*, i.e., an expression of the form $\exists\vec{z}\ \big(\mathit{term}_1(\vec{x},\vec{y},\vec{z}) \wedge \ldots \wedge \mathit{term}_k(\vec{x},\vec{y},\vec{z})\big)$, where each conjunct $\mathit{term}_i(\vec{x},\vec{y},\vec{z})$ is either an expression of the form $w\colon C$ with C being a concept, or wRw' with R being a role, and w, w' are variables from the lists $\vec{x},\vec{y},\vec{z}$, or individual names.

The meaning of the whole expression is that, given a tuple of individuals $\vec{a}$ from $\vec{X}$ as an input, the service S returns as its output the (unordered) set of all tuples of individuals $\vec{b}$ that belong to $\vec{Y}$ and satisfy the condition $\Phi(\vec{a},\vec{b})$.

Let us illustrate this definition by an example adopted from (3). Consider two services S and Q, both with an input of type GeoRegion and an output of type Wine. The service S takes a name of a geographical region (e.g., 'France') as an input and retrieves the list of (names of) wines that are *produced* in this region. The service Q, on the contrary, returns the list of wines that are *sold* in that region. In our framework, they will be described as follows:

$$S = \langle\, x\colon \mathsf{GeoRegion};\ \ y\colon \mathsf{Wine};\ \exists z\,(\, z\colon \mathsf{WineGrower}\ \wedge\ z\,\mathsf{isLocatedIn}\,x\ \wedge\ z\,\mathsf{produces}\,y\,)\,\rangle$$
$$Q = \langle\, x\colon \mathsf{GeoRegion};\ \ y\colon \mathsf{Wine};\ \exists z\,(\, z\colon \mathsf{Shop}\ \wedge\ z\,\mathsf{isLocatedIn}\,x\ \wedge\ z\,\mathsf{sells}\,y\,)\,\rangle$$

Using OWL-S Service Profile, one cannot distinguish between these two services. However, the notion of service matching introduced below takes into account their functionality and hence allows to distinguish between them.

Service matching

Definition 2.2 (Service matching). Given two service descriptions with $|\vec{x}\,| = m = |\vec{z}\,|$ and $|\vec{y}\,| = n = |\vec{w}\,|$:

$$S = \langle \vec{x} : \vec{X};\;\; \vec{y} : \vec{Y};\;\; \Phi(\vec{x}\,, \vec{y}\,) \rangle, \qquad Q = \langle \vec{z} : \vec{Z};\;\; \vec{w} : \vec{W};\;\; \Psi(\vec{z}\,, \vec{w}\,) \rangle,$$

we say that the service S *matches* the request Q w.r.t. the ontology $\mathcal{T}$ if there exist two permutations $\tau\colon \{1, \ldots, m\} \to \{1, \ldots, m\}$ and $\sigma\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that the following two conditions hold:

(i) Applicability: $\mathcal{T} \models X_i \sqsupseteq Z_{\tau(i)}$, for all $i \leqslant m$, i.e., the type of x_i subsumes the type of $z_{\tau(i)}$ w.r.t. the ontology $\mathcal{T}$.
Intuitively: one can map the inputs of S to inputs of Q so that all the user's input data will be acceptable by S.

(ii) Coherence: for any ABox $\mathcal{A}$ and any tuples of individuals $\vec{a}\,, \vec{b}$ in the knowledge base $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A} \rangle$ with $|\vec{a}\,| = m$ and $|\vec{b}\,| = n$, if $\mathcal{KB} \models \vec{a} : \vec{Z}$, then $\mathcal{KB} \models \sigma(\vec{b}\,) : \vec{Y} \wedge \Phi(\tau(\vec{a}\,), \sigma(\vec{b}\,))$ iff $\mathcal{KB} \models \vec{b} : \vec{W} \wedge \Psi(\vec{a}\,, \vec{b}\,)$.
Intuitively: modulo some re-arrangement of the input and output vectors, the services Q and S return the same answers on any input that conforms to the user's request Q.

Theorem 1. *The service matching problem is reducible to subsumption of conjunctive queries w.r.t. an ontology.*

Service composition

Suppose that we are given two services $S = \langle x\colon X; y\colon Y; \Phi(x, y) \rangle$ and $S' = \langle x'\colon X'; y'\colon Y'; \Phi'(x', y') \rangle$. Our task is to formulate the conditions when the composition $S \circ S'$ (to be read as 'first S runs, then S' runs on the output produced by S') is meaningful (i.e., when these services are compatible) and when it matches a user's request $Q = \langle z\colon Z;\; w\colon W;\; \Psi(z, w) \rangle$.

Definition 2.3 (Service composition). A composition of services $S \circ S'$ *matches* a request Q w.r.t. a TBox $\mathcal{T}$ if:

(a) Applicability: $\mathcal{T} \models X \sqsupseteq Z$. This ensures that S accepts all inputs described in the request Q.

(b) Compatibility: $\mathcal{T} \models \forall x, y \big(x\colon Z \wedge \Phi(x, y) \wedge y\colon Y \to y\colon Y' \big)$. The outputs of S (on user's inputs) are accepted by S'.

(c) Coherence: $\mathcal{T} \models \forall x, w \,\big(\Psi(x, w) \wedge w\colon W \;\longleftrightarrow\; \exists y \,\big(\, \Phi(x, y) \wedge y\colon Y \wedge \Phi'(y, w) \,\big) \wedge w\colon Y' \big)$.
On the user's inputs, the application of S and then S' yields the same answers as Q.

Theorem 2. *Matching a composition of services to another service is reducible to query supsumption w.r.t. an ontology.*

Extensions and generalizations

The definition of service matching given above covers only the case when the number of inputs (and outputs) of the service S is equal to that of the service Q. It is almost straightforward to generalize the definition of service matching to the case when user's request Q has "redundant" inputs and/or the service S has "redundant" outputs. There are several possible approaches on how to deal with the remaining cases (when Q has less inputs than S requires, or S has less outputs that Q requires). The framework is also extended to describe services with *structured output* (e.g., linearly ordered or equipped with any other additional structural information) and with *boolean* output. There is an ongoing work on an extension of this framework to describe services whose inputs and outputs are values from concrete domains (i.e., integers, strings, etc.). All these extensions are considered in more details and illustrated by examples in the technical report (1).

Acknowledgements

The research is supported by an EPSRC grant GR/S63182/01, GR/S63168/01 as part of the DynamO project. The authors would like to thank Ian Horrocks, Ulrike Sattler, and Frank Wolter for their help during the research.

References

[1] A. Bovykin and E. Zolin. A Framework for Describing Information Providing Web Services. Tech. report. University of Manchester, 2005. `http://dynamo.man.ac.uk/publications/services_as_queries_TR.pdf`

[2] Terry R. Payne, Massimo Paolucci, and Katia Sycara. Advertising and Matching DAML-S Service Descriptions. Semantic Web Working Symposium (SWWS), 2001.

[3] The DAML Services Coalition. Bringing Semantics to Web Services: The OWL-S Approach. Proc. of the 1st Int. Workshop on Semantic Web Services and Web Process Composition (SWSWPC'2004), July 6-9, 2004, San Diego, California, USA. `http://www.daml.org/services/owl-s/`

Automated Reformulation of Constraint Satisfaction Problems

John Charnley* jwc04@doc.ic.ac.uk
Simon Colton* sgc@doc.ic.ac.uk
Ian Miguel† ianm@dcs.st-and.ac.uk

* Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2AZ
† School of Computer Science, The University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SX

Overview

A constraint satisfaction problem (CSP) consists of a set of variables $\{x_1, x_2, \ldots, x_n\}$, a set of domains of values the variables can take and a set of constraints specifying which values the variables can take simultaneously. A solution to a CSP is an assignment of values to each of the variables from their domains such that no constraints are broken. They find widespread use in science and industry and can be extremely complicated, involving a large number of variables and complex constraints. CSP solvers allow users to specify CSPs in a particular syntax, and then search for solutions to the problem, normally using a configurable search approach. Correctly formulating CSPs is a skilled and time-consuming task. Moreover, once formulated, a CSP problem can take a large amount of processing time to solve. For these reasons, various methods have been devised to improve the effectiveness of CSP solving. One such method is to find additional information about the domain being studied and use this knowledge to reformulate the CSP solver to improve its effectiveness. In particular, when the domain of investigation is described by axioms in first-order logic then it may be possible to derive new theorems from those axioms. Such *implied* theorems are true for all instances of the domain and can therefore be added to the CSP formulation without loss of generality.

(3) used a machine learning system to discover new information about the problem domain of certain CSPs. Their approach was semi-automatic and significant aspects were performed manually. In particular, a skilled constraints programmer created the initial solver models and translated new information into constraints and a skilled mathematician manually reviewed the learning system output. Consequently, although the reformulations provided an improvement in the speed of finding solutions, the gain was offset by the large amount time and expertise required to pre-process the problem.

We have developed a system that fully automates this approach. The system consists of a number of modules, one of which is the HR machine learning system described in (2). HR combines machine learning and automated theorem proving to generate concepts and proven conjectures for a given domain. In the mode of operation that we use, HR starts with the basic axioms of the domain, some examples of objects which satisfy those axioms and some basic concepts of that domain. New concepts are discovered by passing existing concepts through a number of production rules. HR empirically checks the sets of examples which satisfy concept definitions and formulates conjectures whenever there is some relationship between the example sets. HR uses the Otter theorem prover, described in (4), to prove conjectures. Our system also uses the Constraint Logic Programming in Finite Domains library (CLPFD), described in (1), which is part of the SICStus Prolog programming suite. The representation language used by the system is first-order logic, which is understood by HR and Otter. However, CLPFD has its own syntax and so a key part of our system is a bespoke translation suite to convert first-order logic sentences and concepts into CLPFD syntax.

The system is suited to reformulating CSPs for families of related problems with the same axioms expressible in first-order logic e.g. finite algebras. The starting point is the axioms for the family of problems and a template for generating solver models for that domain. The user has two options in creating an initial solver model. They can program the model into the domain template, requiring some knowledge of constraint programming, or they can have the system generate a starting model by automatically translating the domain axioms, using the automated translation routines. In some cases, a combination of both approaches is used. The base solver is used to find solutions to small problem instances, which are used, together with the axioms of the domain, to configure HR for a discovery run.

The proven conjectures HR finds are translated into CLPFD syntax constraints that can be added seamlessly into a solver model, as outlined in §. We apply a generate-and-test approach to finding the best reformulation i.e. the solver model that finds all solutions to the given problem in the shortest time. This will be the base model plus zero or more of the theorems HR has discovered. Individual theorems are assessed by considering their impact upon effectiveness when added to the base model. Testing starts with an initial population of all the theorems discovered by HR. Successive rounds consider combinations of increasing numbers of the best theorems from previous rounds. Testing ends when no improvement can be found, or a user-defined limit is reached, whereupon all reformulations from all rounds are ranked.

Interpretation and Translation of Constraints

First-order logic sentences to be translated into CLPFD syntax are first parsed into nested prolog terms using a Definite Clause Grammar. The translator then recurses through the nested expression creating, in most instances, a new predicate for each level of nesting. We employ methods such as *flattening* and *case-splitting* to improve constraint efficiency. *Flattening* refers to carefully re-translating theorems to simplify them without changing their meaning, for example converting $a * b = c \wedge b * a = c$ to $a * b = b * a$. *Case-splitting* allows us to post simpler, more targeted, constraints representing implication conjectures. For instance, if we take the contra-positive to the conjecture $\forall\, a\, b\; ((a * b = b * a) \rightarrow (a = b))$, then we can case-split between $a = b$ and $a \neq b$, posting the constraint $a * b \neq b * a$ in the latter case.

The main method for translating quantifiers creates predicates to generate sets of value combinations for the quantified variables. Further predicates are used to enforce that a sub-expression holds for all combinations, in the case of universal quantifiers, and for at least one combination, in the case of existential quantifiers. Another method we have used for existential quantification is to represent the quantified variables as additional solution variables and add them to the solver. The solver must assign values to these new variables when finding a solution, which improves efficiency.

Equality expressions require recursive translation, where each sub-expression is first translated in such a way as to produce a result, which are then constrained to be equal. Conjunctions, disjunctions and negations are all translated into the appropriate respective CLPFD syntax in a relatively straightforward manner. Depending upon the domain, some expressions represent variables the solver is trying to allocate values to. For example, in finite algebras, the results of operators, e.g. expressions of the form $a * b$, are the solution variables. If a and b are known then we can identify the exact solution variable affected by the expression. For example, $\forall\, a\, b\; (a * b = b)$ allows us to constrain all variables $(a * b)$ explicitly prior to search. Where nesting occurs, e.g. in $(a * b) * c$, the affected solution variables are only determined when values have been assigned to variables during search. These are translated using the built-in *element* constraint, which is optimised to resolves list values and positions at search time. CLPFD syntax requires that some expressions, e.g. implications, be *reified*, i.e. represented by boolean variables. In the syntax, sub-expressions of reified expressions must also be reified. Thus two translation methods have been developed, with and without reification.

Results

We have used the system to investigate the area of finite algebras. We started with algebras with one operator (*), e.g. quasigroups, and have extended it to algebras with multiple operators e.g.rings. In many cases, the system automatically produced a reformulation that significantly outperformed the model based upon the starting axioms alone. In particular, the system found and used all the theorems discovered in Colton and Miguel (3) using the semi-automatic approach. For example, in the case of QG3 quasigroups, our system automatically reformulated the base model to include the axioms $\forall\, b\, c\, d\; (((b * c = d \wedge c * b = d) \rightarrow (b = c)))$ and $\forall\, b\, c\, d\; (((b * b = c \wedge d * d = c) \rightarrow (b = d)))$, reducing solver time finding examples of size 8 by 68.7%, some 7 hours. In addition, full automation of assessment and translation means there is no longer a need for the users to be particularly skilled in either constraint programming or the domain of investigation. Further, we note that all the useful theorems represent mathematical properties that are interesting in their own right. Consequently, the system is useful in assessing the interestingness of mathematical output from a learning system.

There were a number of algebras where the system did not find an improvement upon the base solver. There are several possible reasons for this. Firstly, it could be that our translation process isn't specifying the most computationally efficient constraints. Alternatively, we may not be detecting useful constraints due to inadequacies of our testing approach, or we may not be finding useful theorems due to the time or configuration of HR. It could simply be the case that no useful derived theorems would be found regardless of how much we looked, in which case the approach could never succeed. Each of these areas can be considered for further work. The system is currently limited to first-order logic, which is insufficiently expressive to cover all CSP problems. Hence, we aim to generalise our approach.

References

[1] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.

[2] S. Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag UK, 2001.

[3] S. Colton and I. Miguel. Constraint generation via automated theory formation. In *Proceedings of the Seventh International Conference on the Principles and Practice of Constraint Programming*, Cyprus, 2001.

[4] W. McCune. The Otter user's guide. Technical report, ANL, 1990.

A Structured Modelling Framework to Capture Symmetry in Circuits

Ashish Darbari*
ashish@comlab.ox.ac.uk
*Oxford University Computing Lab
Wolfson Building
Parks Road
Oxford, OX1 3QD

Introduction

Symbolic trajectory evaluation (6) or STE in short has been successfully used in verification of large industrial sized circuit designs. The data abstraction in STE via Xs is often insufficient to bring about any reasonable reduction in the size of the verification problem. For circuits with large number of state holding elements like random access memories (RAM), content addressable memories (CAM) and caches, these reduction techniques are not adequate. It is also the case that several such circuits exhibit symmetry in their structure, and intuition suggests that exploiting the presence of these symmetries can render significant reductions in the size of the verification problem (3; 5; 4; 2; 7; 1). Our motivation is to be able to perform symmetry based property reduction for STE model checking. In order to do so, we need to be able to have a method of discovering symmetries in circuits. For circuits represented as graphs, a significant proportion of verification effort using symmetry based reduction, goes in *detecting symmetries* in models (5) using subgraph isomorphism. We believe that the problem of symmetry detection is best addressed when symmetries are discovered by using special types in a language (3). In this case the problem of symmetry detection reduces to a problem of type checking.

We have worked on the design and implementation of a type system that is used for designing a structurally symmetric class of circuits. The type system is a way of defining an *abstract data type of models*, rather than attempt of designing an HDL like Verilog or VHDL. We have used this to design several symmetric circuits starting from basic logic gates to multiplexers, comparators, registers, RAMs, CAMs and circuits that involve multiple CAMs. These circuits appear regularly in all modern microprocessor designs.

Design of the framework

Our methodology of designing the framework is based on identifying the following key issues:

1. Propose a sufficiently generic type for circuits.
2. Define the functions that will be used for constructing symmetric circuit blocks.
3. Provide the type judgement rules.
4. Articulate the mathematical definition of structural symmetry.
5. Prove a type soundness theorem, which says that every circuit definition that is well behaved with respect to the typing rules, will have structural symmetry.

A circuit is said to have structural symmetry with respect to a specific set of inputs and outputs when the circuit's behaviour remains unchanged under permutation of those specific set of inputs and outputs. The specific inputs in this case can be termed as symmetric inputs. Every circuit can have some inputs which will be symmetric[1]. There would be some inputs that would not have any role in the symmetry of the circuit, and they are referred to as non-symmetric inputs. Thus a circuit can have the following type:

$$circ : (bool\ list)\ list \to (bool\ list)\ list \to (bool\ list)\ list$$

where the type $(bool\ list)$ denotes the set of values of any one group of wires, and the list of Boolean lists $((bool\ list)\ list)$ denotes the collection of several sets of inputs. The first argument of the circuit type denotes the values of non-symmetric inputs, the second argument denotes the values for symmetric inputs and the third argument the outputs of the circuit.

We use several well known functions on lists, from functional programming, like $head$, $tail$, $append$, map, $map2$, $fold$ and so on, as building blocks for writing higher-order combinators that are used to define symmetric circuit blocks. The type of symmetric circuit blocks is

$$circ : (bool\ list)\ list \to (bool\ list)\ list$$

[1] In case the circuit has no symmetry the set of symmetric inputs will be empty.

The type judgement rules are given by defining a typing predicate SS (symmetry-safe) on the circuit construction combinators, which is defined inductively using nine rules. Arbitrary permutations on lists can be obtained by composing the $swap$ function, which swaps two elements of the list. Symmetry is thus defined in terms of a $swap$ function on lists. A necessary constraint on the symmetric inputs, is that all inputs that are contained in the list of symmetric inputs, should have equal length. This is expressed by the function $CheckLength$.

$$\begin{aligned} Sym\ \ c \quad \triangleq \quad & \forall sym.\ CheckLength\ \ sym\ \ \supset \\ & \qquad \forall i\, j.\ map(swap(i,j))(c\ \ sym) \ = \ c\,(map(swap(i,j))\ \ sym) \end{aligned}$$

Our last step in the design is to prove the type soundness theorem, which in our case states that every circuit block that is well-behaved with respect to the type judgement rules, exhibits symmetry. Thus we prove the following type soundness theorem:

$$\vdash \forall circ.\ SS\ \ circ\ \ \supset\ \ Sym\ \ circ$$

The proof of the soundness theorem relies on proving several properties about lists, and permutations on lists. We used some of the properties about lists, that appear in the list theory in HOL 4, and extended that library for our use, with many other properties that talk about permutations on lists, and functions such as map, $map2$, and $fold$. Once we prove that the type system is correct, we can use the type rules to infer whether or not a given circuit is symmetry safe (SS). We have written tactics that can automate the process of type checking, which takes typically a few seconds. Structured circuit models validated by type checking, are compiled by a combination of HOL functions and an ML program, to a flat netlist usable for simulations in Intel's symbolic simulator Forte.

Acknowledgements

Thanks to my supervisor Tom Melham for providing continual support in this work. This work also benefited in its early stages, by discussions with John O' Leary and Robert Jones at Intel, and Joe Hurd, Tom Newcomb and Ed Smith at Oxford. Hasan Amjad provided useful comments on the abstract.

References

[1] M. Calder and A. Miller. Five ways to use induction and symmetry in the verification of networks of processes by model-checking, 2002.

[2] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry Reductions in Model Checking. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 147–158, London, UK, 1998. Springer-Verlag. ISBN 3-540-64608-6.

[3] C.N. Ip and D.L. Dill. Better Verification through Symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[4] K L. McMillan. A Methodology for Hardware Verification using Compositional Model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[5] Manish Pandey and Randal E. Bryant. Exploiting Symmetry When Verifying Transitor-Level Circuits by symbolic trajectory evaluation. In *CAV*, pages 244–255, 1997.

[6] Carl-Johan H. Seger and Randal E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in System Design: An International Journal*, 6(2):147–189, March 1995.

[7] A. Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702–734, 2004.

Using Proof to Reason about Errors in Ontology Construction

Louise Dennis*
*School of Computer Science and IT, University of Nottingham
Jubilee Campus, Nottingham, NG9 1BB
lad@cs.nott.ac.uk

Introduction

Logic based ontology languages, such as OWL (3), allow the use of automated reasoners to both validate the consistency of the ontology and, in many cases, to compute new subsumption relationships among the classes in the ontology. Efficient tableaux based reasoners such as FaCT (2) can now be embedded in ontology construction tools such as Protégé (4) to perform these tasks. Tutorials on the construction of ontologies (eg. (1)) will often recommend that the deduction of class subsumption relationships which follow from class restrictions be left to such reasoners since they reduce the possibility of user introduced errors.

This paper addresses the question of what happens when a reasoner does not produce the results expected by the user and in particular when it fails to infer the subsumption relationships that the user believes to follow from the class restrictions they have put in place. It commonly believed that the generation of counter-examples in such cases will provide sufficient information for a diagnosis to take place. This paper argues that this is not the case and proposes a technique by which more information could be derived.

A Common Error leading to a Subsumption Failure

(5) identifies two common errors which lead to unexpected classification errors. These errors arise from misunderstandings about the nature of open world reasoning and the difference between sufficient and necessary and sufficient conditions. We illustrate our discussion with an example of the second of these drawn from the above paper.

The user is trying to describe a pizza classification and in particular to give a definition for cheesy pizzas:

$$CheesyPizza \sqsubseteq \exists hasTopping.CheeseTopping$$

This can be understood as asserting that any Cheesy pizza as a Cheese Topping. So having a Cheese Topping is a necessary condition for being a Cheesy Pizza. However the user also "intended" to say that *any* pizza with a Cheese topping is a Cheesy pizza, ie. the condition is both necessary and sufficient.

Having thus defined Cheesy Pizzas the user then attempts to define a Margherita pizza:

$$MargheritaPizza \sqsubseteq (\exists hasTopping.MozzarellaTopping \sqcap \exists hasTopping.TomatoTopping)$$

In this case they have the correct definition since they are trying to say that all Margherita Pizzas have a Mozzarella and a Tomato topping. We have also assumed that $MozzarellaTopping \sqsubseteq CheeseTopping$.

On running their reasoner the user expects that Margherita Pizzas will automatically be classified as Cheesy Pizzas but they are not.

What happens when we try to prove subsumption?

In order to establish a subsumption relationship, $A \sqsubseteq B$, the reasoner will have searched for a refutation proof by attempted to build a tree model of the relationship and demonstrate that no consistent model exists.

In this case it tries to prove $MargheritaPizza \sqsubseteq CheesyPizza$ by negating the goal ($\neg(MargheritaPizza \sqsubseteq CheesyPizza)$) and then transforming it to:

$$MargheritaPizza \sqcap \neg CheesyPizza$$

A consistent model of this statement (our concepts have been abbreviated here for space reasons (eg. $MargheritaPizza$ has become MP) can be built using standard description logic inference rules:

$$\begin{array}{lcc} \mathcal{L}(x) = & \{MP, \neg CP, \exists hT.Moz, & \exists hT.Tom\} \\ & \mid hT & \mid hT \\ & \mathcal{L}(y) = \{Moz, Cheese\} & \mathcal{L}(z) = \{Tom\} \end{array}$$

No inference rules apply to this model and so the statement is judged consistent: the proof process halts without inferring a subsumption relationship between MP and CP.

Following conventional wisdom therefore, an informative counter-model can then be read off and returned to the user to assist with debugging, namely: "x is a Margherita Pizza, which has a Mozzarella Topping, y, (which is also a Cheese Topping) and a Tomato Topping, z. x isn't a Cheesy Pizza" which tells the user little more than the original classification failure did.

Tracing the Ancestry of Properties

Let us annotate this final tree model. In particular we will label all the properties of individual (ie. Mozzarella Topping, Cheese Topping) with a subscript indicating which of the initial two properties they ultimately derive from:

$$\begin{array}{rcc} \mathcal{L}(x) = & \multicolumn{2}{c}{\{MP, \neg CP, \exists hT.Moz_{MP}, \exists hT.Tom_{MP}\}} \\ & \mid hT & \mid hT \\ & \mathcal{L}(y) = \{Moz_{MP}, Cheese_{MP}\} & \mathcal{L}(z) = \{Tom_{MP}\} \end{array}$$

We can see fairly trivially here that there is probably a problem with Cheesy Pizza since we have not made any inferences from it at all. However, more generally, we also observe that we have several leaf nodes (in fact all our leaf nodes in this case) were the properties have only been derived from one of the original properties. In order for a subsumption relationship to be "interesting" then we would expect the clashes which make the model inconsistent to arise between two literals each ultimately derived from a different one of our original classes (otherwise the clash merely tells us that one of our original classes is inconsistent in and of itself).

This observation leads us to search for information that could lead to a clash in one of the available leaf nodes (or one from each model in the cases where we have branching). In general our heuristic suggests that we should expect the culprit for our subsumption failure to lie with a node in our tree model whose children contain only derivations from one of the initial classes not from both. We can then work back from these nodes deducing a general "shape" of missing properties that we hope to unify with, in this example, a necessary condition which could then be a candidate for becoming necessary and sufficient.

References

[1] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe. A practical guide to building OWL ontologies using the Protégé-OWL plugin and CO-ODE tools, edition 1.0. Technical report, University of Manchester, University of Manchester, 2004. Available from http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf.

[2] I. Horrocks. Using an expressive description logic: FaCT or fiction? In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 636–647. Morgan Kaufmann Publishers, San Francisco, California, 1998. URL `download/1998/kr98.pdf`.

[3] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *J. of Web Semantics*, 1(1):7–26, 2003. ISSN 1570-8268. URL `download/2003/HoPH03a.pdf`.

[4] H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen. The Protégé OWL plugin: An open development environment for semantic web applications. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2004. ISBN 3-540-23798-4.

[5] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. Owl pizzas: Practical experience of teaching in owl-dl: Common errors & common patternsn. In E. Motta, N. Shadbolt, A. Stutt, and N. Gibbins, editors, *Engineering Knowledge in the Age of the Semantic Web, 14th International Conference, EKAW 2004*, volume 3257 of *Lecture Notes Computer Science*, pages 63–81. Singer, 2004.

XOR Fragments of Temporal Logic

C. Dixon*, M. Fisher* and B. Konev*
*Department of Computer Science
The University of Liverpool, Liverpool L69 7ZF, UK
{clare,michael,konev}@csc.liv.ac.uk
http://www.csc.liv.ac.uk/research/logics/

Introduction

Propositional linear-time temporal logic (PTL) has been a popular logic for reasoning about reactive systems. However, as deciding the complexity of satisfiability for PTL is PSPACE-complete, applying deductive verification to this logic appears impractical. Yet, although the model checking problem for PTL is also PSPACE-complete, practical model checking tools have been developed. An explanation for this in (2) is that most formulae tackled in practice fall in to one of a number of efficient subfragments of PTL. Here we look at efficient subfragments of PTL from a deductive (resolution) based perspective. Our starting point defines a particular fragment of PTL allowing XOR clauses. This fragment can be used to represent Büchi Automata.

PTL-X$_\mathcal{A}$ and SNFX$_\mathcal{A}$

We define a new logic, PTL-X$_\mathcal{A}$, which is essentially a subfragment of PTL. We assume the usual syntax and semantics for PTL (see for example (4)) but allow a new XOR operator defined as follows. For some model σ and moment in time $i \in \mathbb{N}$ the XOR operator is defined as follows.

$$(\sigma, i) \models \varphi_1 \oplus \varphi_2 \oplus \ldots \oplus \varphi_m \quad \text{iff there is exactly one } 1 \leqslant j \leqslant m \text{ such that } (\sigma, i) \models \varphi_j.$$

The new logic, PTL-X$_\mathcal{A}$, comprises exactly those clauses that can be represented in a particular normal form (SNFX$_\mathcal{A}$). Note, in this logic, the set of propositions, $\mathcal{P}$, is partitioned into two disjoint sets, $\mathcal{S}$ and $\mathcal{L}$. We can use these to represent *states* and *labels* if we use this logic to translate Büchi Automata into SNFX$_\mathcal{A}$. SNFX$_\mathcal{A}$ is of the general form $\square \bigwedge_i A_i$ where each A_i must be one of the following.

$$\begin{array}{lcll lcll}
\textbf{start} & \Rightarrow & \bigvee_i q_i & (\textit{initial}\text{ clause}) & \textbf{true} & \Rightarrow & \Diamond \bigvee_i q_i & (\textit{sometime}\text{ clause}) \\
(q_k \wedge l_j) & \Rightarrow & \bigcirc \bigvee_i q_i & (\textit{step}\text{ clause}) & \textbf{true} & \Rightarrow & q_1 \oplus q_2 \oplus \ldots \oplus q_n & (\textit{XOR-}\mathcal{S}\text{ clause}) \\
\textbf{true} & \Rightarrow & R_c & (\textit{universal}\text{ clause}) & \textbf{true} & \Rightarrow & l_1 \oplus l_2 \oplus \ldots \oplus l_m & (\textit{XOR-}\mathcal{L}\text{ clause})
\end{array}$$

In SNFX$_\mathcal{A}$, at most one *sometime* clause and at most one *initial* clause is allowed. Further, for every q_k, l_j such that $q_k \in \mathcal{S}$ and $l_j \in \mathcal{L}$ there is at most one clause of the form $q_k \wedge l_j \Rightarrow \bigcirc \bigvee_i q_i$ in the clause set. $\mathcal{S}$ must equal $\{q_1, q_2, \ldots, q_n\}$ and $\mathcal{L}$ must equal $\{l_1, l_2, \ldots, l_m\}$. Thus, all elements of $\mathcal{S}$ and $\mathcal{L}$ occur within some XOR clause. R_c must be one of $\neg q_i$, or $(\neg q_i \vee \neg l_j)$, where $q_i \in \mathcal{S}$ and $l_j \in \mathcal{L}$.

Clausal Temporal Resolution for SNFX$_\mathcal{A}$

Next we consider resolution rules for sets of SNFX$_\mathcal{A}$ clauses. The resolution rules are split into four groups initial resolution; step resolution; hyper resolution and temporal resolution. *Initial resolution* involves resolving an initial clause with a universal clause.

$$\boxed{\mathsf{IRES}} \quad \begin{array}{rcl}
\textbf{start} & \Rightarrow & Q \vee q_i \\
\textbf{true} & \Rightarrow & \neg q_i \\ \hline
\textbf{start} & \Rightarrow & Q
\end{array}$$

Step resolution resolves step clauses with universal clauses (SRES1) or derives additional universal clauses from contradictions obtained in the next moment (SRES2).

$$\boxed{\mathsf{SRES1}} \quad \begin{array}{rcl}
q_i \wedge l_j & \Rightarrow & \bigcirc (Q \vee q_k) \\
\textbf{true} & \Rightarrow & \neg q_k \\ \hline
q_i \wedge l_j & \Rightarrow & \bigcirc Q
\end{array} \qquad \boxed{\mathsf{SRES2}} \quad \begin{array}{rcl}
q_i \wedge l_j & \Rightarrow & \bigcirc \textbf{false} \\ \hline
\textbf{true} & \Rightarrow & \neg q_i \vee \neg l_j
\end{array}$$

Hyper XOR Resolution takes several universal clauses relating to the negation of a proposition in $\mathcal{S}$ with the XOR-$\mathcal{L}$ clause (HRES). *Temporal Resolution:* (TRES) since there is only one sometime clause which is of a simple form (i.e. it has

true on the left hand side) we can use a simplified version of the standard (4) step resolution rule, defined in (1).

$$\boxed{\mathsf{HRES}}\quad \begin{array}{rcl} \mathbf{true} & \Rightarrow & \neg q_k \vee \neg l_1 \\ \ldots & \Rightarrow & \ldots \\ \mathbf{true} & \Rightarrow & \neg q_k \vee \neg l_m \\ \mathbf{true} & \Rightarrow & l_1 \oplus \ldots \oplus l_m \\ \hline \mathbf{true} & \Rightarrow & \neg q_k \end{array} \qquad \boxed{\mathsf{TRES}}\quad \begin{array}{rcl} Q & \Rightarrow & \Box \bigwedge_i \neg q_i \\ \mathbf{true} & \Rightarrow & \Diamond \bigvee_i q_i \\ \hline \mathbf{true} & \Rightarrow & \neg Q \end{array}$$

The soundness and completeness of this set of resolution rules is shown in the full paper. Due to the limited number of clauses that can be derived, the overall complexity of a naive procedure applying these rules is polynomial in the number of propositions. See (3) for more details.

Conclusions and Future Work

The fragment we have defined here, whilst being efficient to decide, is very restricted. However, we can use this to efficiently represent and reason about Büchi Automata. The automata-theoretic view underlying PTL model checking is that both the set of paths and the property to be checked can be captured using finite automata over infinite sequences. Thus can we use the results from this work to generate $\mathrm{SNFX}_{\mathcal{A}}$ clauses from the automata representing the set of paths and automata representing the (negation of) the property to be checked and to efficiently check emptiness via temporal resolution? We aim to extend this fragment to consider more general fragments allowing more than two XOR sets and a less restrictive form of the clauses and to extend to first-order temporal logic.

References

[1] A. Degtyarev, M. Fisher, and B. Konev. A simplified clausal resolution procedure for propositional linear-time temporal logic. In U. Egly and C. G. Fermüller, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of *Lecture Notes in Computer Science*, pages 85–99. Springer-Verlag, July 30–August 1 2002. ISBN 3-540-43929-3.

[2] S. Demri and P. Schnoebelen. The Complexity of Propositional Linear Temporal Logic in Simple Cases. *Information and Computation*, 174(1):84–103, 2002.

[3] C. Dixon, M. Fisher, and B. Konev. Is there a Future for Deductive Temporal Verification? (Submitted), 2006.

[4] M. Fisher, C. Dixon, and M. Peim. Clausal Temporal Resolution. *ACM Transactions on Computational Logic*, 2(1): 12–56, January 2001. ISSN 1529-3785.

Tool Support for Names and Dependencies in Deductive Synthesis

Lucas Dixon[⋆]
lucas.dixon@ed.ac.uk
[⋆]Informatics
University of Edinburgh

Introduction

Deductive synthesis has many interesting applications including the construction of verified programs ((3)), the conjecturing of lemmas and generalisations ((2)), and the synthesis of specialised induction schemes ((1)). However, the lack of tool-support makes research into its applications difficult. In particular, existing interactive theorem provers, such as Isabelle ((4)), lack appropriate support for managing complex proof attempts with variables shared between different goals and assumptions.

In this note we examine the machinery needed to manage names and dependencies for interactive deductive synthesis. We note the difficulties inherent in the task as well as those particular to the Isabelle proof assistant.

Deductive Synthesis

Deductive synthesis refers to proofs in which an object is synthesised. Typically, this occurs in constructive proof attempts containing existentially quantified variables. For example, the synthesis of a sorting algorithm can be considered as a proof of the following statement:

$$\forall l : list(\mathbb{N}).\ \exists s : list(\mathbb{N}).\ ord(s) \wedge perm(l, s)$$

where ord is a predicate that tests whether a list is ordered and $perm$ is one that checks whether the first list is a permutation of the second. A constructive proof will provide a specifc witness for the variable s. This will be a function on l, which we will call $sort(l)$. For example, synthesising quicksort would yield the following definition:

$$\begin{aligned} sort([]) &= [] \\ sort(h\#t) &= sort(leq(h,t)) \mathbin{@} [h] \mathbin{@} sort(more(h,t)) \end{aligned}$$

where $leq(h,t)$ is a function that results in the elements of t less than or equal to h, and $more(h,t)$ results in those strictly greater than h. The @ function is the infix append.

Deductive Synthesis in Isabelle

Isabelle is a generic higher order theorem prover. It supports meta-variables which can be (partially) instantiated by higher order unification. Their freshness, dependencies, and any flex-flex constraints which relating variables in higher order unification, are managed by Isabelle. However, this management of meta-variables is not ideal. In particular, applying tactics often results in seemingly arbitrary renaming of variables. Furthermore, the flex-flex constraints are hidden and cannot easily be separated into object level goals. This makes it difficult, even in small proofs, to tell which meta-variables correspond with each other. Another problem with the management of variables and their instantiation is that once a proof is completed, it is not necessarily possible to derive how the initial meta-variable was instantiated. This information is lost unless the proof is recorded, which slows down the proof process.

Another important difficulty when working in Isabelle's ML interface is that subgoals and assumptions, which are stored in lists, cannot be referred to consistently. Whenever a tactic is applied, assumptions and goals may change position. The Isabelle/Isar language for writing proofs does address this issue: it allows assumptions and intermediate results to be given fixed names within a scope. However, proof scripts written in Isar may not contain meta-variables. Thus the instantiations must be known before hand and cannot be built up incrementally. The reason for this limitation is that meta-variables in Isabelle are only managed across a single proof state. The ability to name assumptions and intermediate results is provided by breaking the proof attempt into several sub-states.

Features Needed for Interactive Deductive Synthesis

We propose that an Isar-like interface that supports meta variables should be developed. In particular, we suggest that the following features are needed to make an effective interface for deductive synthesis:

Named results and assumptions: It should be possible to refer to assumptions, goals, and intermediate lemmas, with user-defined names. This is to support the application of assumptions and lemmas to goals in an unambiguous manner. For example, the synthesis of induction schemes, as proposed in ((1)), involves several concurrent proof attempts with many intermediate lemmas and goals. Without suitable names it is difficult for the user to guide the proof. Furthermore, using explicit names makes the proof script for the synthesis more robust to changes in definitions and tactics.

Nice fresh invariant names: New assumptions, goal, lemmas and variables are all introduced during synthesis. Furthermore, they must all be referred to at various times by the user. This requires them to be given names. Such names must be fresh to ensure soundness as well as to avoid ambiguity. Furthermore, because of the large number of names, they should be as readable as possible and performing a proof step should not change them. Doing so will make the proof process very confusing as well as making it difficult for the user to refer to these objects.

Management of Variable-Goal Dependencies: When a variable is instantiated in a proof step, then the variable should be updated in all other goals in which it occurs. This also requires tracking the dependencies of goals on variables and updating them appropriately when new variables are introduced.

Management of Variable-Instantiations: During the synthesis process, variables are instantiated and new variables are introduced. The user wants to know the final instantiation of the initial variables which are being synthesised. Thus initial variable instantiations should not be thrown away, even after all terms that refer to them have been instantiated; instead they must be managed and updated appropriately. This is important as the user wants to see both the intermediate and final instantiations and make sure that it meets their expectation. This can help the user guide the synthesis process.

Conclusions and Further Work

We have presented a brief analysis of the tool-support that we believe is needed to facilitate names and variable dependencies for interactive deductive synthesis. We believe that this is an important step in the automation of deductive synthesis. We are developing a extension to Isabelle that implements these features and are now examining their use for the implementation of proof critics and for the deductive synthesis of grid workflows.

References

[1] A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing induction rules for deductive synthesis proofs. In S. Allen, J. Crossley, K.K. Lau, and I Poernomo, editors, *CLASE-05 Workshop*, pages 4–18. University of Edinburgh, 2005.

[2] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[3] Z. Manna and R.J. Waldinger. A deductive approach to program synthesis. *Journal of Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[4] L.C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, 1994.

A Web Search System for Theorems

He Cheng*
*Graduate School of Science and Technology, Nihon University
Tokyo 101-8308, Japan
jeffhe04@yahoo.co.jp

Hideo Suzuki†
†Tokyo Institute, Polytechnic University
Tokyo 187-0035, Japan
hd_suzu@yahoo.co.jp

Hidetsune Kobayashi*
hd_koba@yahoo.co.jp

Abstract

This paper presents a web search system for theorems. The system has the following features: (1) It supports quick search by web browsers. (2) It provides a plentiful theorem database for search. (3) By the system, theorems can be found by various ways.

Introduction

Theorem is the basis of all theorem provers as the provers need existing theorems to prove new ones. Usually, thousands of theorems exist in the provers for powerful proof. However, they also make it hard for users to memorize so many of them. Therefore, it is necessary to build a flexible and general system for theorem finding.

Like the idea proposed by Suzuki and Funato (3), in this paper, a work of search system for theorems is presented to meet the need of looking for theorems. It can be accessed by web browsers, and its data is based on a plentiful database of theorems, which are extracted from a well-known generic theorem prover, Isabelle (1). The most important one is that the system supports various ways to search.

Overview

The structure of the system is shown in **Fig.1**. The work flow of the system is: At first, theorems in Isabelle are collected and kept as records in file by **ML Interface** written in ML (2). Then they are inserted into a database constructed by postgreSQL system. Finally, using **PHP Interface** and the services provided by Apache, users of web server or remote clients can search the theorems through web browsers.

Implementation

According to the work flow above, the implementation of the system could be divided into the following three parts:

1. Theorem extraction: Theorems are extracted from internal Isabelle by newly developed ML functions. The functions produce the theorem records. Each of the records contains seven kinds of information about theorem, such as name, content, variables, the number of variables (variable number for short), variable order, kind, and constants.

2. Theorem management: As the theorem records are kept in text files, they can be easily put into the database by sql commands like *copy*. Some indexes are created in the database to increase the search's efficiency.

3. Theorem search: The search is performed in the PHP interface. The interface not only provides spaces for receiving user queries but also includes several sub-processes, such as communicating with the database, dealing with the queries and rewriting the variables in results.

Experiment

The prover Isabelle only supports searching theorems by the constants contained in them. Compared with Isabelle, the system provides more ways to search quickly in the database containing 18910 theorem records in total, which are obtained from Isabelle's all logics by theorem extraction. The ways include search by theorem name, by variable number, by variable order, by theorem content or by their combination.

A search test could indicate that search by the system is flexible and fast.**Table.1** shows the results of the test with five available search methods. The variables in *thm-ex* are enclosed by "?" for variable extraction from the query. Replacing

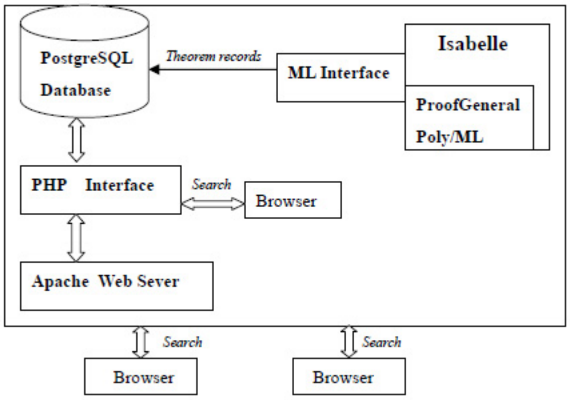

Figure 1: The search system's architecture

Table 1: Results of a theorem search test. The search times are got from a 2.4GHz Intel PC with 640MB of RAM. The query *thm-ex* is ”$?m? \ =?n?; ?m? <?n? ==>?R?; ?n? <?m? ==>?R? ==>?R?$”.

No	Search Method	Query	Theorems	Time(s)
1	By theorem name	set	1860	0.205
2	By variable number	3	2742	0.162
3	By variable order	0,1,0,1,2,1,0,2,2	6	0.021
4	By theorem content	*thm-ex*	4	0.010
5	Compound search	0,1,0,1,2,1,0,2,2;<	4	0.023

them with ”x”,”y” and ”P” respectively does not affect the results except that the variables are rewritten with the new ones. The compound search returned the same 4 theorems to that of the fourth search. Its query is a combination of the variable order ”0,1,0,1,2,1,0,2,2” and the constant ”<” as a part of theorem content. The advantage of this way is that it saves the users the time in entering all the variables and constants. The last three ways can greatly reduce the search space and raise the accuracy of the search.

Conclusions

This paper describes a work of realizing a web search system for theorems. The work enables users to find theorems efficiently. It can also be considered as a general theorem search system for theorem proving because the system presents results in web pages and has a general database based on all the theorems from the generic prover, Isabelle.

For future work, we plan to connect theorem proving with the system directly and realize distributed parallel theorem proving based on the system.

References

[1] L. C. Paulson. *Isabelle–A Generic Theorem Prover*. Springer Verlag, 1994.

[2] L. C. Paulson. *ML for the working programmer (2nd edition)*. Cambridge University Press, 1996.

[3] H. Suzuki and H. Funato. On the implementation of a proof database. *Journal of JSSAC*, 10(2), 2003.

A Tableau Metamodel For Description Logics

Nenad Krdžavac[⋆†]
[⋆]Faculty of Electrical Engineering, University of Belgrade
Bulevar Kralja Aleksandra 73, 11 000 Belgrade, Serbia and Montenegro
nenadkr@galeb.etf.bg.ac.yu

Vladan Devedžić[†]
[†]FON - School of Business Administration, Department of Information Technology, University of Belgrade
POB 52 Jove Ilića 154 11000 Belgrade, Serbia and Montenegro
devedzic@etf.bg.ac.yu

Abstract

The paper presents a method for implementation of a tableau algorithm for description logics based on tableau metamodel as a part of the architecture of a description logics reasoner. The solution is based on the Model Driven Architecture (MDA) methodology. The design pattern called Composite is used in the tableau metamodel.

Introduction

The basic notions in description logics (DLs) are concepts (unary predicates) and roles (binary predicates) (1). One of the most important constructive properties of DLs is their reasoning services. Reasoning services in DLs are based on the tableau algorithm (1). The tableau algorithm tries to prove the satisfiability of concept term C, by demonstrating a model in which C can be satisfied (4). Some publicly available implementations of DL reasoners (4; 3) can reason with ontologies, but the authors of those reasoners did not describe the model of reasoning algorithms using advanced model engineering techniques and current software engineering standards.
The objective of this paper is to describe the tableau metamodel for a description logics reasoner using the Model Driven Architecture (MDA) approach (5). The paper discusses some benefits of using the MDA methodology to implementation that kind of algorithms. We suggest possible applications of that methodology to implementation of the tableau algorithm in other theorem provers. We also propose an application of the implemented tableau algorithm to intelligent systems (we exemplify it by the AIR system (2)) and to intelligent analysis of students' solutions in Web-based intelligent tutoring systems (ITS).

The Tableau Metamodel

Our DL reasoner is based on the DL metamodel, proposed by the OMG consortium (www.omg.org) (6), and on the tableau metamodel (Fig. 1). Implementation of an MDA-based tableau algorithm for DLs includes the following steps:

1. Building the tableau metamodel for description logics;
2. Implementing a model repository for the tableau metamodel;
3. Implementing the reasoning algorithms using model transformation.

The tableau metamodel can be built using Unified Modeling Language (UML) (8). The tableau algorithm uses a tree (T) to represent the model being constructed (4), hence the Composite design pattern (7) can be used in describing the tableau metamodel.
The tableau metamodel can be logically divided in two parts. The first part make the metaclasses that represent concept terms in DLs, and correspond to metaclasses in the proposed DLs metamodel (6). Some of them ones are:

1. *Formula* that represents common description logics terms;
2. *AtomicFormula* that represents atomic building blocks in DLs, like roles and atomic concepts;
3. *SubFormula*

These three metaclasses make an instance of the Composite design pattern (Fig.1).The second part makes the metaclasses and relationships among them that represent the model (in the context of DLs definition) that is constructed according to expansion rules (4). The metaclass *Individual* represents all nodes in such a model. Every node corresponds to some set of

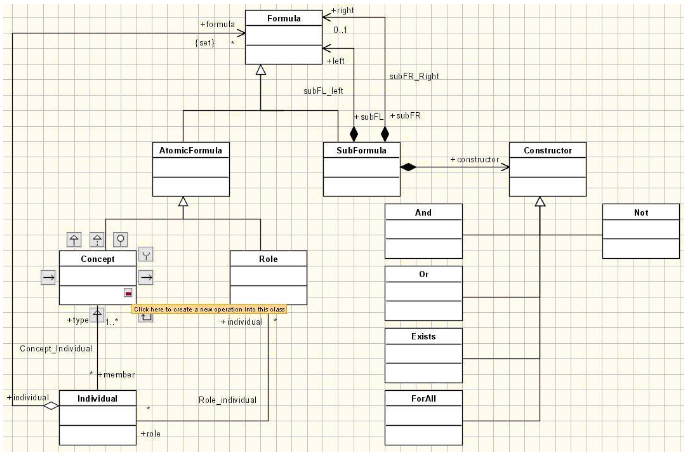

Figure 1: A Tableau Metamodel

concept terms. This is represented by the association (aggregation) between the metaclasses Formula and Individual and an OCL constraint set (Fig. 1) at the end of the association. In some cases the set may be empty. Every role connects at least two individuals. It is represented by the association between the metaclasses Role and Individual.
The metamodel supports only the ALC description logic. The support is provided by the metaclasses that represent logical constructors like Or, Not, And, ForAll, Exists.

Conclusions and Future Works

The paper presents a method for implementation of reasoning algorithms for description logics. We proposed a tableau metamodel for description logics. Such an implementation methodology for the reasoner is flexible in practical applications of the reasoner in today's intelligent metamodeling frameworks or in Web-based intelligent tutoring systems. Our future work will be focused on extension of the tableau metamodel to support DLs with concrete domains and functional dependencies. We will test our reasoner in various Semantic Web applications.

References

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi , P. Patel-Schneider, *The Description Logic Handbook-Theory, Implementation and Application*, Cambridge University Press, 2003.

[2] D. Djuric, D. Gasevic, V. Damjanovic, AIR-A Platform for Intelligent Systems, *In Proceedings of AIAI 2004: First IFIP International Conference on Artificial Intelligence Applications and Innovations*, Toulouse France, 2004.

[3] V. Haarslev, R. Moller, RACER Systems Description, *Lecture Notice in Computer Science*, 2001.

[4] I. Horrocks, *Optimising Tableaux Decision Procedures for Description Logics*, PhD Thesis, University of Manchester, 1997.

[5] Meta Object Facility (MOF) Specification, v1.4, [Online]. Available: http://www.omg.org/docs/formal/02-04-03.pdf

[6] *Ontology Definition Metamodel*, Preliminary Revised Submission to OMG RFP ad/2003-03-40 1, 2004. [Online]. Available: http://codip.grci.com/odm/draft

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable of Ebject-Oriented Software, Addison-Wesley, Reading, MA, 1995, ISBN: 0201633612

[8] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide , Addison-Wesley, 1998, ISBN: 0201571684

A Real Arithmetic Test Suite for Theorem Provers

Olga Lightfoot*
*DCS, Queen Mary
University of London, London E1 4NS
olgo@dcs.qmul.ac.uk

Abstract

This paper discusses the construction of the test suite for evaluating the capabilities of an interactive theorem prover to deal with arithmetic on real numbers. Although linear arithmetic operations and equality reasoning are quite well understood and automated, handling non-linear operations, especially involving transcendental functions, is still a non-trivial task. We analyse several classes of problems and show what results can be deduced about the system after such tests on the example of PVS.

Introduction

Formal verification of both hardware and software often requires fairly complex manipulations of mathematical formulae and, with the rising popularity of formal methods, task automation is increasingly important. An industrial application normally deals with a high volume of expressions, so one would like as many statements or intermediate steps in the proofs to be discharged automatically by the proof tool in use.

One of the advantages offered by theorem provers is that they operate on the exact mathematical expression, rather than on a discrete approximation offered by other means of computer implementation. The difference in real number construction methods, as well as the different logical paradigms behind the systems themselves, result in the variety of ways in which the existing higher order logic theorem provers handle certain classes of expressions involving real numbers.

Independent testing procedures comparing the performance of the competing products on a range of problems are common practice in both software and hardware industries, but although a number of benchmarks for first order logic theorem provers exist, such as TPTP (1) for automated theorem provers, a common or accepted benchmark across the range of higher order theorem proving which would focus on real number arithmetic is still missing. Such a test suite would be highly beneficial in deciding which proof engine to choose based on the ease of manipulation in a given domain, rather than just existing expertise of the future users. It would also serve as a motivation for refining the proof tools further. We propose a test suite that examines the capabilities of a theorem prover to deal with the real arithmetic, and discuss the results obtained after running it on PVS (Prototype Verification System).

Test suite and PVS representation

A number of things are important to keep in mind when designing a test suite, such as the breadth of the testing area, modularity in the complexity of the test cases and their relevance to the general area. Since it is generally impossible to conduct empirical testing on all possible inputs, we have to construct a finite set of tests verifying how well the system in question satisfies our expectations.

Here we report on the ongoing work, and so far the test suite consists of over one hundred problems drawn from various sources, all of which have been run in PVS. Our goal is to accumulate a representative body of examples that will illustrate standard algebra and analysis problems that high-school and first year undergraduate students are expected to be able to solve and of the functions which are more likely to be used in formal methods applications.

We have included illustrative examples of the main identity transformations for both algebraic and transcendental expressions, i.e.:

- Algebraic equalities: starting from linear and progressing to higher degrees on rational and irrational expressions
- Algebraic inequalities: linear and non-linear on rational and irrational expressions
- Series
- Transcendental functions: trigonometric and logarithmic functions, exponentials

The examples include most of the main functions that are essential for more complex expression constructions (such as factorials for the binomial expressions) and, as we are starting from the most basic and commonly used building blocks, more complicated statements can be added later. We do not intend to tackle proving function properties such as checking continuity or differentiability in this work, unless such statements are missing from the current libraries and are required for a successful completion of the proofs of the identity transformations and inequalities in our problem library.

PVS is based on the sequent calculus logical framework with the specifications formulated in terms of classical higher order logic. Most of the types, functions and operations we need, such as real and natural numbers, are predefined in the PVS specification language. One feature of PVS which has affected the way the examples are represented in the suite is its restriction to total functions. We have to define the domain where the expression is valid before starting to formally prove anything about it, and in PVS we have two choices: either restrict the domain of the function by specifying the type or include it in the statement as a precondition. We have chosen to use preconditions for its greater transparency and potential ease for inter-format portability, not to mention avoiding the awkwardness of having too many expression-sensitive variable types.

One of the main goals of this work is to test how much of the proof could be done with automated decision and semi-decision procedures, so, naturally, we first attempt to use the existing prover procedures and strategies. In PVS, `grind` is the most powerful command, invoking all the decision procedures and rewrite rules known to the prover. However, as the difficulty of the proof increases, it is easy to see how relying on the default rewriting mechanisms fails, so we have to start using other commands and packages, such as `field` and `manip` which were specifically designed to ease handling of the algebraic expressions, or explicitly guiding the prover with lemmas.

For the analysis the results of executing the test suite we have decided to consider the timing, length of the proof and level of automation, as well as the level of ingenuity required from the user if the automated tools fail to complete the proof.

Choosing the strategy influences the length and direction of the proof dramatically. Once the automated facilities of the prover are exhausted without achieving the desired effect, we have to start guiding the tool along to get the right result. One approach is to give the prover clues and perform some transformations for it without explicitly calling the intermediate step lemmas, hoping that most of the correctness proofs for such substitutions will be either discharged by the automated decision and rewriting procedures or with relatively simple proofs, another — actively deploy the lemmas and use only the auxiliary statements which have been priorly proved in a given tool. When we compare the proofs of the same problem achieved with different approaches in PVS, their length can vary dramatically.

So, if we accept the proof length as one of the criteria of assessment of the efficiency of a theorem prover and expand the testing to other theorem provers, we have to consider whether the same kind of strategies are used in different proof tools, besides the obvious differences in the level of automation of certain operations in competing systems.

Conclusions and future work

The length of the proof and its complexity often depends on the way the problem is presented to the proof tool. If a lot of facts are proved about one representation in a particular system, it is advantageous to make the translation into the most explored notation. At the moment we represent the test problems in two formats: traditional mathematical notation and PVS syntax. The natural continuation of this work is to derive a common machine-readable representation format for the problems and run it on other theorem proving engines, afther which a more challenging problems could be added with more information on the performance across the field. Our choice of representing function domains as preconditions allows a relatively easy conversion to the notation of another prover which admits partial functions, since the preconditions could be simply stripped from the problem formulation. A few choices for derivation of a common machine-readable notation are Prolog format (like in TPTP), MathML or MathWeb notation.

References

[1] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21 (2):177–203, 1998.

Dynamic Refinement in Knowledge Management

Fiona McNeill

Diverging Knowledge

A perennial problem of knowledge management is that large databases or ontologies with multiple users will suffer from contents divergence (1; 2). They must be maintained by updating their contents as it becomes apparent during use that parts of it are insufficient, redundant, incorrect or inappropriately represented. Additionally, decentralised information can be altered by individual users and thus become out of sync with other copies of that information. This issue is not only unavoidable, but also, in many situations, desirable, because it allows knowledge sources to be adapted to specific circumstances without the need to create an entirely knowledge source. However, such differences cause problems in interactions between systems or agents with access to different versions of the knowledge.

There are two main approaches to this situation. One is to view conflicting knowledge as problematic and to attempt to resolve conflicts to reintroduce consistency. The other is to embrace the need for divergence and to focus on how to allow agents or systems with conflicting knowledge to interact successfully with each other despite the conflicts: for example, by building bridges or maps between the conflicting information. Additionally, the differences between conflicting pieces of information may be purely semantic (i.e., conflicting facts), or they may also be syntactic (i.e., the underlying representation does not match).

Dynamic Ontology Refinement

We investigated this situation in the context of agent ontology mismatch in (4). Much work in the field of ontology mapping is concerned with how entirely different representations can be mapped by linking similar concepts. In such approaches, it is normally necessary to have access to large parts of both of the ontologies to be mapped, and to do this mapping off-line. However, our dynamic ontology refinement system (ORS) is designed to address the problem of small mismatches between agents with largely similar ontologies which have diverged, in the manner described above. Such an approach allows us to avoid making strong assumptions such as the assumption that it is possible to gain complete access to other agents' ontologies, or that it is acceptable to do such mapping off-line. Instead, interaction proceeds by assuming that the ontologies are the same, and conflicts between the ontologies are only investigated when they cause problems in the agent communication process. The fact that large amounts of the ontology are shared then gives a basis for conducting discussions about what the root of the conflict might be, and allows agents to attempt to understand one another's conflicting knowledge in terms of their shared knowledge. Hence only the relevant ontological mismatches are investigated, and this can be done dynamically during runtime.

We investigated this problem for agents using KIF (3) ontologies, which is a full first-order representation, and thus provides a rich catalogue of potential mismatches. The KIF representation was analysed to determine in what ways it was possible to alter it, and hence how differently updated ontological objects could differ. Diagnostic algorithms were then devised to determine which of these possibilities had occurred. It was assumed that multiple mismatches were distinct and could be diagnosed serially in order to simplify the problem; situations where this is not a valid assumption cannot usually be accurately diagnosed and patched by ORS. The system was then evaluated by observing existing first-order ontologies for which different versions were available, and comparing the mismatches between these versions with the mismatches that ORS could identify and patch. 40% of all these mismatches could be identified and patched by ORS, and for the vast majority of those that could not, this failure was due to limitations in the scope of the project (for example, the avoidance of interdependent multiple mismatches and differences in representation between the test ontologies and KIF), rather than to theoretical limitations in the approach.

OpenKnowledge

We are currently investigating how our dynamic refinement techniques could be used in the EU OpenKnowledge project (5). The purpose of the project is to attempt to provide a framework to enable a Semantic Web like system to become open, in the sense that anyone may join and the cost of individual participation is very low. Such semantic based systems have thus far been very costly to join because of the semantic commitments that must be made: being precise about semantics for sophisticated components is prohibitively expensive and the cost of ensuring an individual, absolute semantics for a component rises rapidly as more participants take part. It is clear that the success of the World Wide Web stems largely from the fact that it is extremely easy for anyone to join, and if the Semantic Web is going to become successful, it will need to mirror this approach. This means, in practice, that mismatched semantics have to be dealt with dynamically during runtime where necessary, rather than forcing users to deal with these problems before they can join the system.

Of the many challenges associated with such a task, one of these is that syntax is dynamic. Even though there may some recognised representation for certain semantics, they will be constantly updated and altered, both centrally and by individuals or groups, so that small mismatches begin to appear that can lead to problems. In the broader context of the OpenKnowledge system, it could exist not only between ontologies, but also between protocols, service definitions, and so on.

Conclusion

Dynamic refinement has been demonstrated to be useful for facilitating communication between agents with mismatched ontologies (4), and our work in the OpenKnowledge project will help to determine how widely applicable these ideas are.

References

[1] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 311–322, New York, NY, USA, 1987. ACM Press.

[2] Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multi-perspective specifications. In *European Software Engineering Conference*, pages 84–99, 1993.

[3] M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Stanford, CA, USA, 1992.

[4] Fiona McNeill. *Dynamic Ontology Refinement*. PhD thesis, School of Informatics, University of Edinburgh, 2006.

[5] OpenKnowledge. http://www.openk.org/, 2006.

Normal Forms for Modal Logics

Cláudia Nalon*

*Departamento de Ciência da Computação
Universidade de Brasília
Caixa Postal 4466
Brasília - DF - Brazil
nalon@unb.br

Abstract

We are currently investigating how to extend, in an uniform way, the clausal resolution procedure for a variety of normal modal logics. As the proof method is clausal, the formula we are checking for satisfiability is firstly transformed into its normal form before applying any of the possibly many resolution rules. Efficient translation, in the sense that this transformation results in fewer and/or shorter clauses, is essential for practical use of the method. Here, we briefly discuss some of the issues related to translation into the normal form.

Logics of belief, epistemic, deontic, and temporal logics, just to cite a few, are some of several modal logics that have been used in computer science for reasoning about situations as, for instance, the specification of distributed (3) and multi-agent systems (7). Given a logical specification, an automated tool, for instance, a theorem prover, can then be used for verification. However, such specification may be written by using a combination of different logics, in order to model the different aspects of a particular situation. When the combination is given by the *fusion* of logical systems, that is, when the components are independently axiomatisable, proofs can be relatively easily obtained by combining the provers for each language. Nevertheless, this may require the use of tools which are based on different implementations (e.g. different input languages) or, worse, on different approaches (e.g. partially based on translation × partialy based on the modal language, resolution × tableau), making this task harder.

We propose an uniform way to deal with theorem proving for a variety of *normal propositional modal logics*, that is, logics in which the schema $\Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$ (**K**), where $\Box$ is the modality for *necessity* and φ and ψ are formulae, is valid. The systems we are investigating are mono-modal normal logics based on the axioms given in Table 2, where $\Diamond$ is the modality for *possibility*. The proof method for each logic is resolution-based and our approach is clausal: in order to prove that a formula, φ, is valid, we first transform its negation, $\neg\varphi$, into a clausal normal form. After translation, resolution rules are applied until a contradiction is found (and the original formula is valid) or no new clauses are generated (and the original formula is not valid).

$$\begin{array}{rl}
\mathbf{K}: & \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi) \\
\mathbf{T}: & \Box\varphi \Rightarrow \varphi \\
\mathbf{D}: & \Box\varphi \Rightarrow \Diamond\varphi \\
\mathbf{4}: & \Box\varphi \Rightarrow \Box\Box\varphi \\
\mathbf{5}: & \Diamond\varphi \Rightarrow \Box\Diamond\varphi \\
\mathbf{B}: & \Diamond\Box\varphi \Rightarrow \varphi
\end{array}$$

Table 2: Axioms

We note that there are some differences between (purely) propositional resolution and that designed to deal with modal logics. Firstly, in propositional logic there is only one rule to apply to the set of clauses, whilst we often need several rules for dealing with modal logics. Transformation into clausal form, as in the propositional case, is carried out by performing classical style rewriting, simplification, and renaming (6), a technique which may avoid combinatorial explosion on the size of the formula by replacing complex suformulae by new symbols, whose meaning are linked to the formula that they are replacing. The simplification step, however, is more complex, as it depends on the particular logic we are dealing with. For instance, in epistemic logic, as the schema $\Box\Box\varphi \Rightarrow \Box\varphi$ (because of **T**) is valid, nested modal operators can be replaced by only one modality. We cannot use this simplification rule, though, if that schema does not hold in the particular logic we are considering.

Efficient translation is crucial for practical use of the resolution method. Although most of our work is based on that of (2), the normal form differs slightly (where l_i are literals, i.e., propositions or their negations): clauses are separated into literal clauses (disjunctions of literals), positive modal clauses (an implication as $l_1 \Rightarrow \Box l_2$), and negative modal clauses (an implication as $l_1 \Rightarrow \neg\Box l_2$). This further separation allows a better designing of strategies to guide the search for a proof when applying the resolution method (4). For instance, complete strategies for (purely) propositional logic could be used when the parents are literal clauses. When the parents are modal clauses, an obvious choice would be implementing a set of support strategy, where the positive modal clauses are in the usable set and the negative modal clauses are in the set of support (or vice-versa). Similar strategies could be applied for the other modal rules, reducing the space search and improving efficiency.

Also, we are currently investigating how the ideas of selective renaming, given in (1) and (5), could be used here in order to produce less and/or shorter clauses. Besides, we are also investigating how transforming the formula into its prenex or anti-prenex form, prior to translation into the normal form, could affect the resulting (translated) formula.

Acknowledgements

This work was supported by CNPq grants CT-INFO 506598/04-7 and Universal 47171/2004-0.

References

[1] Thierry Boy de la Tour. Minimizing the number of clauses by renaming. In Mark E. Stickel, editor, *Proc. 10th International Conference on Automated Deduction, Kaiserslautern*, LNCS 449, pages 558–572. Springer-Verlag, July 1990.

[2] Clare Dixon and Michael Fisher. Resolution-based proof for multi-modal temporal logics of knowledge. In S. Goodwin and A. Trudel, editors, *Proceedings of the Seventh International Workshop on Temporal Representation and Reasoning (TIME'00)*, pages 69–78, Cape Breton, Nova Scotia, Canada, July 2000. IEEE Computer Society Press. ISBN 0-7695-0756-5.

[3] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[4] Cláudia Nalon. *Resolution for Synchrony and No Learning*. PhD thesis, University of Liverpool, March 2004.

[5] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science B.V., 2001.

[6] David A. Plaisted and Steven A. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Logic and Computation*, 2:293–304, 1986.

[7] Anand S. Rao and Michael P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3): 293–342, 1998.

Mechanical Verification of Recursive Procedures Manipulating Pointers using Separation Logic

Viorel Preoteasa*

*Department of Computer Science
Åbo Akademi University and
Turku Centre for Computer Science
DataCity, Lemminkäisenkatu 14A, Turku 20520, Finland
viorel.preoteasa@abo.fi

Abstract

Using a predicate transformers semantics of programs, we introduce statements for heap operations and separation logic operators for specifying programs that manipulate pointers. We prove consistent Hoare total correctness rules for pointer manipulating statements according to the predicate transformer semantics. We prove the frame rule in the context of a programming language with recursive procedures with value and result parameters and local variables, where program variables and addresses can store values of any type of the theorem prover. The theory, including the proofs, is implemented in the theorem prover PVS.

Introduction

Separation logic (3; 5) is a powerful tool for proving correctness of imperative programs that manipulate pointers. However, without theorem prover support, such tasks are unfeasible. By employing Isabelle/HOL theorem prover and separation logic, Weber (4) implements relatively complete Hoare logics for a simple while programming language extended with heap operations. Nevertheless, his implementation does not treat (recursive) procedures and local variables.

In this paper, we introduce a predicate transformer semantics for imperative programs with pointers and define separation logic constructs. Based on this semantics, we prove Hoare total correctness rules for heap operations (new, dispose, lookup, and update). Our work is implemented in the theorem prover PVS and it is based on a previous formalization (1) of Refinement Calculus (2) with recursive procedures.

Even if possible, we have chosen not to implement address arithmetic (3) in our calculus. Unlike approaches presented by Reynolds (3) or Weber (4), where memory addresses are integers and can store only integer values, our work allows addresses to store values of any type available in the theorem prover.

The main contribution of this work is the formal proof of the frame rule in the context of a programming language with recursive procedures with value and result parameters and local variables, where program variables and addresses can store values of any type of the theorem prover.

The semantic of the programming language

Let Σ be the state space. Here, predicates, denoted Pred, are the functions from $\Sigma \to \mathsf{bool}$. MTran is the type of all monotonic functions from Pred to Pred. Programs are modeled as elements of MTran. If $S : \mathsf{MTran}$ and $p : \mathsf{Pred}$, then $S.p : \mathsf{Pred}$ are all states from which the execution of S terminates in a state satisfying the postcondition p. The program *sequential composition* denoted $S\ ;\ T$ is modeled by the functional composition of monotonic predicate transformers, i.e. $(S\ ;\ T).p = S.(T.p)$. If α and β are predicates and S is a program, then a *Hoare triple*, denoted $\alpha\ \{\!|\ S\ |\!\}\ \beta$ is true if and only if $\alpha \subseteq S.\beta$.

We assume three infinite disjoint types: $\mathsf{variable}$, $\mathsf{address}$, and $\mathsf{constant}$ and two constants: $\mathsf{alloc} \in \mathsf{variable}$, $\mathsf{nil} \in \mathsf{constant}$; alloc stores the set of allocated addresses during the execution of a program and nil represents the null address. The type $\mathsf{location}$ denotes the disjoint union of $\mathsf{variable}$ and $\mathsf{address}$, and for all $x \in \mathsf{location}$, $\mathsf{T}.x$ is the type of x. For example the type of alloc is the power set of $\mathsf{address}$. The type $\mathsf{V}.X = \{x \in \mathsf{variable} \mid \mathsf{T}.x = X\}$ represents the program variables of type X. An element of $\mathsf{B}.X = \{x \in \mathsf{address} \mid \mathsf{T}.x = X\} \cup \{\mathsf{nil}\}$ is either nil or is an address that can store an element of type X. For example, the type of the program variables of type addresses to natural numbers is $\mathsf{V}.(\mathsf{B}.\mathsf{nat})$.

Heap operations and separation logic

All separation logic operators and heap operations can be introduced based on the elements presented so far. For example, separation conjuction of two predicates, $\alpha, \beta \in \mathsf{Pred}$, denoted $\alpha * \beta$, is true in a state σ if there exists a subset A of $\mathsf{alloc}.\sigma$

such that α is true in the state $\sigma[\mathsf{alloc} := A]$ and β is true in $\sigma[\mathsf{alloc} := \mathsf{alloc}.\sigma - A]$, where $\sigma[x := a]$ represents the state in which the value of the location x is changed to a. The predicate emp is true in all states where alloc is empty. The statement Dispose(a) removes the address a from alloc if a is an allocated address, or it does not terminate otherwise.

Recursive procedures and frame rule

The type of procedures with parameters from A, denoted Proc.A, is $A \to \mathsf{MTran}$. A call to a procedure $P \in \mathsf{Proc}.A$ with the actual parameter $a : A$ is the program $P.a$. Every monotonic function F from Proc.A to Proc.A defines a recursive procedure $P \in \mathsf{Proc}.A$, $P = \mu\, F$, where $\mu\, F$ is the least fixpoint of F.

Let ptree be the record type with the components label : nat, left : B.ptree, and right : B.ptree. The predicate tree.a, defined for all addresses $a \in$ B.ptree, is true in a state σ if the heap of σ (allocated addresses together with their values) represents a binary tree with the root at the address a. The predicate tree.a is true only in those states where the heap represents exactly the tree with the root in a, and there are no extra addresses allocated.

A procedure which disposes a tree from memory, denoted DisposeTree, is an element from Proc.(V.(B.ptree)). If $u \in$ V.(B.ptree), then DisposeTree.u disposes the tree stored in u and sets u to nil. The specification of this procedure is:

$$\mathsf{tree}.u \ \{\!|\ \mathsf{DisposeTree}.u\ |\!\}\ \mathsf{emp} \wedge u = \mathsf{nil} \qquad (1)$$

This correctness statement asserts that if the heap contains only a tree with the root in u, after calling DisposeTree.u the heap is empty and the value of u is nil. However, we cannot use this property in contexts where the heap contains other addresses in addition to the ones specified by tree.u. For example, in a recursive implementation of DisposeTree, the right subtree is in the heap if we first dispose the left subtree. We would like to derive a property like:

$$\alpha * \mathsf{tree}.u \ \{\!|\ \mathsf{DisposeTree}.u\ |\!\}\ \alpha \wedge u = \mathsf{nil} \qquad (2)$$

for all predicates α which does not contain u free. This can be achieved using the frame rule.

If A is a type of procedure parameters and $X \subseteq A \to \mathsf{Pred}$, then we define

$$\mathsf{Proc}_X.A = \{P \in \mathsf{Proc}.A \mid \forall \alpha \in X,\ \forall q \in A \to \mathsf{Pred} \bullet \alpha * P.q \subseteq P.(\alpha * q)\}$$

For the procedure DisposeTree the set X is $\{\alpha : \mathsf{V}.(\mathsf{B.ptree}) \to \mathsf{Pred} \mid \alpha.u$ does not contain u free$\}$.

Theorem 1. *If* $p, q : A \to \mathsf{Pred}$ *and* $body : \mathsf{Proc}.A \to \mathsf{Proc}.A$ *is monotonic, then the following Hoare rule holds*

$$\frac{\begin{array}{l}(\forall w : W,\ P : \mathsf{Proc}_X.A \bullet p \wedge t < w\ \{\!|\ P\ |\!\}\ q \Rightarrow p \wedge t = w\ \{\!|\ body.P\ |\!\}\ q)\\ \wedge\ (\forall P : \mathsf{Proc}_X.A \bullet \mathsf{Proc}_X.A.(body.P))\end{array}}{(p\ \{\!|\ \mu\, body\ |\!\}\ q)\ \wedge\ \mathsf{Proc}_X.A.(\mu\, body)}$$

When proving a recursive procedure we can assume a stronger property (like (2)), and prove a weaker property (like (1)). When using the procedure corectness statement in proving other programs, we can also use a stronger property (like (2)).

References

[1] R.J. Back and V. Preoteasa. An algebraic treatment of procedure refinement to support mechanical verification. *Formal Aspects of Computing*, 17:69 – 90, May 2005.

[2] R.J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.

[3] J. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*, 2000.

[4] Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004. ISBN 3-540-23024-6.

[5] H. Yang and P.W. O'Hearn. A semantic basis for local reasoning. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes In Computer Science*, pages 402–416, London, UK, 2002. Springer-Verlag. ISBN 3-540-43366-X.

Boolean Function Simplification Applied in Code Simplification

Mihis Andreea-Diana*
*"Babes-Bolyai" University, Cluj-Napoca, Romania
mihis@cs.ubbcluj.ro

Abstract

Boolean function simplification is used to realize simpler combinatorial circuits. But this method can be used to simplify complicated conditions from code; this condition simplification process can be automated. My further work concentrates on building a instrument capable to simplify Boolean functions.

In the software development process, certainly some conditions will appear, sometimes these conditions are independent one by another and sometimes they are composed. In the case in which they are composed, a simplification of these composed condition is necessary, to simplify in this way the code.

Why don't we use here a Boolean function simplification method? Certainly, we can replace every simple condition with a x_i, where i is a natural number between 1 and n, and n is the total number of conditions. Because every condition will have the value 1 ($TRUE$) if it is satisfied and 0 ($FALSE$) if it is not satisfied, we can consider their composition as a function $f(x_1, x_2, \ldots, x_n)$ (which will have the result 0 or 1).

So, the code simplification problem can be reduced to a Boolean function simplification. And this process of simplification can be automated, using an application (or a mini CASE) to which I am working now.

To automatic simplify a Boolean Function, Quinne (- McClusky) method is the best choice to be implemented. Because this method can be applied only to functions written in normal canonical disjunctive form, first, the function must be transformed into a disjunctive form. This process can be implemented on a computer too, following the steps:

1. if there are, $\rightarrow$ and $\leftrightarrow$, they must be eliminated, using the following rules:
   $U \leftrightarrow V$ is $(U \rightarrow V) \wedge (U \rightarrow V)$
   $U \rightarrow V$ is $\neg U \vee V$
2. if there are hole expressions negated, DeMorgan formulas are applied (eventually in generalized form):
   $\neg(U \wedge V)$ is $\neg U \vee \neg V$
   $\neg(U \vee V)$ is $\neg U \wedge \neg V$
3. if is possible, the following distributive rule is applied (eventually in generalized form):
   $U \wedge (V \vee Z)$ is $(U \wedge V) \vee (U \wedge Z)$

In this moment, the function is in a disjunctive form, but not certainly in normal canonical disjunctive form. Anyway, because for Quinne method we need the function's support, the next step is to write it's support, namely the set of groups of values that the variables must have, such that the function will have value 1. In other words, the powers of variables are written, where 1 is the power of x_i, 0 is the power of $\neg x_i$ and if a variable does not appear in a conjunctive term, is replaced by 0 and 1. This process too is easier implemented.

In my doctoral stage, among other things, I want to realize a mini CASE that implements the above process. It will help me in my research, but it will be free to be used by students and other researchers.

References

[1] Donald E. Knuth. *Art of Computer Programming, Fundamental Algorithms*, 3rd. Ed. Addison Wesley, Reading MA, 1997.

[2] Lawrence C. Paulson. *Logic and Proof* University of Cambridge, 2000.

Translations to Propositional Satisfiability

Juan Antonio Navarro-Pérez*
*Supervisor: Andrei Voronkov, Adviser: Renate Schmidt

The University of Manchester
School of Computer Science
Oxford Rd. Manchester M13 9PL, UK.
navarroj@cs.manchester.ac.uk

Introduction

Propositional satisfiability (SAT) solvers have recently been found as a competitive approach in many industrial applications. In component verification, for example, one is given a formal description of a system (usually a finite state machine) and a property that is to be checked against the system. Clarke et al. (1) proposed a method, *bounded model checking*, where both the finite state machine and the property are encoded into a propositional logic formula. The formula is then fed into a satisfiability solver and, if a satisfying model is found, it can be used as a counterexample to show an execution trace in which the system fails to satisfy the required property.

Dramatic improvements in SAT solver technology (4) and the availability of efficient implementations (3) have been continuously improving the applicability of this approach in industrial settings. However, researchers have also observed that a very relevant decision is how to encode or translate the problem into a form that is suitable for use by a satisfiability solver. Even a slight modification in syntax of the resulting formula could change the search space explored by the solver in a rather unpredictable way. In this short report we present an overview of our experiences translating bounded model checking problems into satisfiability solving, outlining both ideas that seem promising and others that did not prove to be very successful.

An economic clause normal form translation

In order to simplify its design and implementation, modern satisfiability solvers typically work on formulae in *clause normal form* (CNF) so that arbitrary propositional formulae first need to be translated into this format. Tseitin (6) proposed a translation that works by introducing *definitions* that map each subformula of the original expression to a new atom, e.g. $x \leftrightarrow (a_1 \wedge \cdots \wedge a_n)$. The newly introduced atoms can be used instead of the corresponding subformulae while their definitions, which are simpler flat formulae, can be easily translated to CNF. Plaisted and Greenbaum (5) further improve this idea showing that, by considering the *polarity* of subformulae (i.e. how it is affected by the scope of negations), it is sometimes enough to include one implication of the definition, e.g. $x \rightarrow (a_1 \wedge \cdots \wedge a_n)$ only.

It has been noted, however, that the translation tends to introduce many unnecessary atoms. One of the simplifications of the SATELITE preprocessor (2), for example, is specifically targeted to identify atoms introduced by Tseitin's translations and 'undo' their definition. This motivated our interest to design a translation which is still polynomial, but trying to introduce as few new atoms as possible. Some of our main ideas follow:

- We consider logic connectives with multiple arity. In particular equivalence and exclusive disjunction are replaced by more general *parity constraints* (i.e. a parity constraint is true if an odd/even number of their operands is true).
- The formula is translated to a *negation normal form* first, where the negation connective is only applied to atoms, but parity connectives are still allowed anywhere. This simplifies the computation of *polarity*: every formula has positive polarity, except for subformulae under the scope of parity constraints which have polarity zero.
- Subformulae with polarity zero are translated doing a *full renaming* as Tseitin does. Other subformulae can be more carefully translated, adding only one atom for each conjunction, using the idea that a *half-definition* $x \rightarrow F$, where F is already in CNF, can be translated without adding any more atoms (simply append $\neg x$ to each clause in F).
- *Parity clauses*, which are parity constraints applied to literals only, are left untouched until the very end of the translation. At this point, parity clauses of length 2 and 3 can be translated using a standard clausification. Longer parity clauses, of length n, can be split into $n - 2$ parity clauses of length 3 by adding $n - 3$ new atoms.

An important difference of our implementation, with respect to common approaches (e.g. 1), is that we do not use subformula sharing. This is under the rationale that, in problems from real life applications, it is unlikely to expect large blocks of identical subformulae to share. If two large identical formulae were needed in the description of a system, the human writing such description would probably recognise it and emulate the 'sharing' by introducing a definition of a new atom himself. This can be observed, for instance, in typical benchmarks of model checking problems where system descriptions consist mainly of definitions.

In some preliminary experimental results we did obtained some significant improvements compared to the translation provided by the BMC tool of (1). For a particular example (a safety property on a barrel shifter), our translation halved in average both the total number of clauses and variables of the formula produced, allowing speedups of solving time by an order of magnitude. However, more experimentation is needed in order to give any conclusive results.

A SAT solver with direct support of parity clauses

During the development of our translation we observed that, potentially, most atoms are introduced while clausifying parity clauses. It then made sense to implement a solver with direct support of parity clauses. Moreover, common implementation tricks such as the *watched literals scheme* (4) could be easily adapted to work with parity clauses.

We implemented this idea in a version of MINISAT (3) with support of custom constraints but, unfortunately, it did not quite work as expected. First we observed that problems from typical applications do contain large amounts of parity clauses, but most of them are of size 2 or 3 which do not introduce new atoms when clausified anyway. Moreover the possible advantages of the approach, such as having a more compact representation and a reduced number of literals to watch, turned out to be negligible for this short parity clauses.

Another problem we faced is that there was no obvious way to write a formula such as $x \to (a \oplus b)$,[1] which are often produced in half-definitions of the translation, using clauses and parity clauses only. One approach was to clausify the right hand side and loose any advantage of having support of such parity constraints. The other approach was to ignore the polarity optimisation in such cases and write a single parity clause $(\neg x \oplus a \oplus b)$. This, however, dramatically decreased the effectiveness of the solver, making it explore a considerably larger search space in all our experiments.

Conclusions

This is currently work in progress. We have been trying many ideas and exploring possible paths to get better representations of problems stemming from real life applications and solve them with propositional satisfiability methodologies. Some of this ideas have proved useful, and some others have not. We have yet a few ideas to try in order to get concise propositional representations of formulae and, in our immediate future work, we also plan to study the applicability of alternative translations into decidable fragments of first order logic.

References

[1] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, 2001. ISSN 0925-9856.

[2] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. 8th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*. Springer, 2005.

[3] Niklas Eén and Niklas Sörensson. MINISAT — a SAT solver with conflict-clause minimization. Poster presented at the SAT 2005 Competition, 2005.

[4] David G. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85: 112–133, February 2005. Logic in Computer Science Column.

[5] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3): 293–304, 1986. ISSN 0747-7171.

[6] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part II*, 1968.

[1] We use $\oplus$ to denote the odd parity connective.

An Hypersequent Calculus for Łukasiewicz Logic without the Merge Rule

Robert Rothenberg[⋆]

[⋆]School of Computer Science, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SX, UK
rr@dcs.st-and.ac.uk

Abstract

The hypersequent system **GŁ** for Łukasiewicz logic presented in (4) is discussed. A variant system is derived which eliminates the weakening rules and combines all of the axioms and the merge rule into a single form of axiom, resulting in a system which appears more convenient for root-first proof search.

Introduction

Hypersequent systems, first presented in (7) and (1), and surveyed in (2) and (3), are a generalisation of Gentzen-style sequent systems which is suitable for some non-classical logics. Hypersequents are multisets of sequents (called "components")

$$\Gamma_1 \Rightarrow \Delta_1 \mid \Gamma_2 \Rightarrow \Delta_2 \mid \ \ldots \ \mid \Gamma_n \Rightarrow \Delta_n$$

where the "pipe" operator | is a meta-level disjunction. An hypersequent is true in some interpretation if at least one of the components is true in that interpretation. This generalisation allows hypersequent systems to use one rule for what may be multiple rules in their sequent counterparts.

In addition to the standard sequent rules, hypersequent calculi have "external" structural rules which affect multiple components. A rule such as external contraction can be used to eliminate duplicate components, for instance. Shuffling rules are a class of external structural rules which combine or exchange information from multiple components, such as the communication rule[1] in the system **HG** for Gödel-Dummett logic from (3),

$$\frac{G_1 \mid \Gamma_1, \Gamma_1' \Rightarrow A_1 \quad G_2 \mid \Gamma_2, \Gamma_2' \Rightarrow A_2}{G_1 \mid G_2 \mid \Gamma_1, \Gamma_2' \Rightarrow A_2 \mid \Gamma_2, \Gamma_1' \Rightarrow A_1} \ (Com)$$

Naïve proof search with shuffling rules must deal with an exponential number of cases, which is impractical for hypersequents with many components or large components. The complexity of implementing root-first proof search on these rules for hypersequent calculi can outweigh advantages these calculi may provide over their sequent counterparts.

The Hypersequent Calculi GŁ and GŁ2

The hypersequent system **GŁ**, introduced in (4) for infinite-valued Łukasiewicz logic contains the following axioms and rules:

$$(\Lambda) \ \Rightarrow \qquad (ID)\ A \Rightarrow A \qquad (\bot)\ \bot \Rightarrow A$$

$$\frac{G \mid \Gamma, B \Rightarrow A, \Delta \mid \Gamma \Rightarrow \Delta}{G \mid \Gamma, A \supset B \Rightarrow \Delta} \ (L\supset) \qquad \frac{G \mid \Gamma, A \Rightarrow B, \Delta \quad G \mid \Gamma \Rightarrow \Delta}{G \mid \Gamma \Rightarrow A \supset B, \Delta} \ (R\supset)$$

$$\frac{G \mid \Gamma \Rightarrow \Delta}{G \mid \Gamma, A \Rightarrow \Delta} \ (LW) \qquad \frac{G}{G \mid \Gamma \Rightarrow \Delta} \ (EW) \qquad \frac{G \mid \Gamma \Rightarrow \Delta \mid \Gamma \Rightarrow \Delta}{G \mid \Gamma \Rightarrow \Delta} \ (EC)$$

$$\frac{G \mid \Gamma_1 \Rightarrow \Delta_1 \quad G \mid \Gamma_2 \Rightarrow \Delta_2}{G \mid \Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2} \ (M) \qquad \frac{G \mid \Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2}{G \mid \Gamma_1 \Rightarrow \Delta_1 \mid \Gamma_2 \Rightarrow \Delta_2} \ (S)$$

This calculus **GŁ** has the sub-formula property, but the presence of both external weakening (EW) and contraction (EC) rules is problematic. It can be shown that the weakening rules (LW and EW) can be absorbed into the axioms for a new system **GŁ2**, in a manner similar to that used in (8) for the system **G2**. However, the presence of the merge rule (M)—a simplified shuffling rule—makes proof search expensive.

[1]The Roman letter G is used in hypersequent rules to represent side components of 0 or more sequents.

Eliminating the Merge Rule from GŁ2

An hypersequent is *balanced* iff it has the form

$$G \mid \Gamma', \overbrace{\bot, \ldots, \bot}^{n}, \Delta' \Rightarrow A_1, \ldots, A_n, \Delta' \text{ where } n \geq 0$$

It can be shown that the axioms of **GŁ2** are balanced. *Imbalance* is defined as maximal distance in the number of edges in a normalised derivation tree from from a node to a balanced hypersequent.

It can be shown that merge inferences in normalised derivations can be permuted towards the leaves (thus allowing a reduction in the size of the search space), so that their premisses are either axioms or other merges. That is, the imbalance of the conclusions of merges is reduced.

From this, the system **GŁ2′** is derived which eliminates the weakening rules and combines the axioms and the merge rule into a single form of axiom:

Theorem 1. . The system **GŁ2′** with the following rules

$$(Axiom)\ G \mid \Gamma', \overbrace{\bot, \ldots, \bot}^{n}, \Delta' \Rightarrow A_1, \ldots, A_n, \Delta' \text{ where } n \geq 0$$

$$\frac{G \mid \Gamma, B \Rightarrow A, \Delta \mid \Gamma \Rightarrow \Delta}{G \mid \Gamma, A \supset B \Rightarrow \Delta}\ (L\supset) \qquad \frac{G \mid \Gamma, A \Rightarrow B, \Delta \quad G \mid \Gamma \Rightarrow \Delta}{G \mid \Gamma \Rightarrow A \supset B, \Delta}\ (R\supset)$$

$$\frac{G \mid \Gamma \Rightarrow \Delta \mid \Gamma \Rightarrow \Delta}{G \mid \Gamma \Rightarrow \Delta}\ (EC) \qquad \frac{G \mid \Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2}{G \mid \Gamma_1 \Rightarrow \Delta_1 \mid \Gamma_2 \Rightarrow \Delta_2}\ (S)$$

is sound and complete for infinite-valued Łukasiewicz logic.

It is noteworthy that all rules in **GŁ2′** except S are invertible.

Experimental work is in progress. A Prolog implementation of **GŁ2′** is available.

Future Work

Other hypersequent and sequent systems where the merge rule is admissible will be investigated in order to determine whether a similar technique of permuting the rule is applicable. Another area of research is to investigate whether similar techniques can be applied to other shuffling rules.

Heuristics relating to the external contraction and split rules will be investigated, with the possibility of deriving a terminating system which eliminates those rules. The system would be similar to the labelled system **GŁ$_l$** from (6), which consists of a single form of axiom and logical rules.

References

[1] A. Avron. A constructive analysis of RM. *Journal of Symbolic Logic*, 52(4):939–951, December 1987.

[2] A. Avron. The method of hypersequents in the proof theory of propositional non-classical logics. In W. Hodges, M. Hyland, C. Steinhorn, and J. Truss, editors, *Logic: Foundations to Applications*. Oxford Science Publications, 1996.

[3] M. Baaz, A. Ciabattoni, and C.G. Fermüller. Hypersequent calculi for Gödel logics—a survey. *J. Log. Comput.*, 13 (6):835–861, 2003.

[4] A. Ciabattoni and G. Metcalfe. Bounded Łukasiewicz logics. In M. Cialdea Mayer and F. Pirri, editors, *TABLEAUX 2003*, volume 2796 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2003.

[5] Jan Łukasiewicz. Philosophische bemerkungen zu mehrwertigen Systemen des Aussagenkalküls (Philosophical remarks on many-valued systems of propositional logics). In Storrs McCall, editor, *Polish Logic 1920-1939*. Clarendon Press, Oxford, 1967.

[6] George Metcalfe, Nicola Olivetti, and Dov Gabbay. Sequent and Hypersequent Calculi for Abelian and Łukasiewicz Logics. *ACM Transactions on Computational Logic*, 6(3):578–613, 2005.

[7] G. Pottinger. Uniform cut-free formulations of T, S4 and S5 (abstract). *Journal of Symbolic Logic*, 48(3):900, 1983.

[8] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge U.P., 1996.

Solvability with Resolution of Problems in the Bernays-Schönfinkel Class

Renate Schmidt⋆ Ullrich Hustadt†

⋆University of Manchester
schmidt@cs.man.ac.uk

†University of Liverpool
U.Hustadt@csc.liv.ac.uk

Abstract

We present a number of results on the solvability and complexity of problems in the Bernays-Schönfinkel (BS) class. We focus on resolution methods and show how the BS class and some natural subclasses can be decided by general-purpose resolution procedures. The decision procedures can be easily implemented in current resolution theorem provers for first-order logic and initial tests are encouraging.

The Bernays-Schönfinkel class (BS) is one of the well-known quantifier prefix classes which are decidable. Formulae in the BS class are conjunctions of function-free and equality-free formulae of the form $\exists^*\forall^*\psi$, where ψ is free of quantifiers. Bernays and Schönfinkel (1) showed that this class is decidable. The complexity of its satisfiability problem is NEXPTIME-complete (2). Plaisted (6) proved the EXPTlME-completeness of the satisfiability problem of Horn problems in the BS class. The class of Horn BS formulae is of interest because Datalog (without negation, equality and numeric operations) is a subclass of the Horn BS class and is in fact satisfiability equivalent to Horn BS. That the BS class is decidable is not difficult to see. Since the existential quantifiers never occur below any universal quantifier Smullyan type tableau systems will always terminate. This also means that the Herbrand model of a satisfiable BS formula is always finite. It is therefore not difficult to see that most if not all variations of instantiation-based methods: first-order Davis Putnam, hypertableau, the model evolution calculus, and the disconnection calculus are decision procedures for the BS class. It is shown in Leitsch (5) that while hyperresolution decides the Horn fragment of the BS class, the BS class is not decidable by either hyperresolution or any A-ordering refinement of resolution. There are people who believe that resolution cannot in general decide the BS class, and it seems to be indeed the case that no general-purpose resolution decision procedures have been described in the literature for BS. However, descriptions of decision procedures using resolution *can* be found in the literature. These procedures exploit the Herbrand theorem and the fact that the size of Herbrand model is determined by the number of occurrences of existential quantifers in a BS formula. Since the Herbrand universe is finite there is an obvious reduction of any problem in BS to propositional logic, and then we can use resolution. This provides a method of deciding BS with resolution. Computing the set of all ground instances of a problem is of course potentially very expensive, but it is also possible to use a partial instantiation method. The idea here is to use partial instantiation to reduce any problem in BS into a problem of another class which *is* decidable by resolution. Such a method is described in Leitsch (5). The algorithm transforms a problem in BS into the intersection of BS and the syntactic class PVD (3), which is decidable by hyperresolution.

We have developed new, general-purpose decision procedures using resolution for both the class the BS formulae and the class of BS clauses. The methods do not use full or partial instantiation. Instead the idea is to use resolution with some minor adjustments in the interpretation of the problem formulation. In particular, we exploit that any first-order formula, or clause set, can be effectively transformed into a set of range-restricted clauses. A clause is range-restricted if all the variables of the positive part of the clause also occur in the negative part. We can prove:

Theorem 1. *The class of range-restricted BS clauses is decidable by hyperresolution and all refinements.*

This then give us effective and natural resolution decision procedures for BS formulae and BS clauses.

Theorem 2.
1. *There is a linear, satisfiability equivalence preserving transformation of any BS formula, and any set of BS clauses, into a set of range-restricted BS clauses.*
2. *Hyperresolution and all refinements can decide the class of BS formulae and the class of BS clauses.*

The developed methods are easy to realise in any first-order resolution theorem prover. The implementation effort is modest, and since the methods are sound and complete for first-order logic we are optimistic that implementors of resolution theorem provers can be persuaded to integrate the methods into their systems. Results of initial tests with our partial implementation in MSPASS (9), especially on BS problems, are encouraging.

The BS class has further natural subclasses for which results on the decidability and complexity can be obtained.

Theorem 3. *Hyperresolution (with or without factoring) decides Horn BS clauses in optimal time (EXPTIME).*

Corollary 0.1. All refinements of hyperresolution decide Horn BS clauses in optimal time (EXPTIME).

Another interesting subclass is the class of modal BS formulae (or clauses). This class arose from our investigations and efforts to develop of resolution-based decision procedures for modal logics and description logics. Modal BS is equivalent to the basic path logic into which the basic modal logic K and the description logic $\mathcal{ALC}$ can be embedded. A formula belongs to modal BS if it has the form $\varphi = \exists\overline{x}\forall\overline{y}\,\psi$ where $\overline{x}, \overline{y} \subseteq Z_m$ and ψ is defined by:

$$\psi \longrightarrow P(z_1, \ldots, z_i) \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi$$

It follows from results in Schmidt (7, 8) that this class is decidable by unrefined resolution with condensing and therefore also *any* refinement resolution with condensing, including hyperresolution with condensing.

Theorem 4. *All refinements of resolution with condensing decide the modal BS class.*

Results in Georgieva et al. (4) imply that hyperresolution with splitting can decide this class in polynomial space. Since satisfiability in the modal logic K and the description logic $\mathcal{ALC}$ is PSPACE-complete, this gives us an optimal procedure.

Theorem 5. *Hyperresolution with splitting can decide the modal BS class with optimal complexity (PSPACE).*

References

[1] P. Bernays and M. Schönfinkel. Zum Entscheidungsproblem der mathematischen Logik. *Math. Annalen*, 99:342–372, 1928.

[2] L. Denenberg and H. R. Lewis. Logical syntax and computational complexity. In *Computation and Proof Theory*, volume 1104 of *Lecture Notes in Mathematics*, pages 101–115. Springer, 1984.

[3] C. Fermüller, A. Leitsch, T. Tammet, and N. Zamov. *Resolution Method for the Decision Problem*, volume 679 of *Lecture Notes in Computer Science*. Springer, 1993.

[4] L. Georgieva, U. Hustadt, and R. A. Schmidt. Computational space efficiency and minimal model generation for guarded formulae. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 85–99. Springer, 2001.

[5] A. Leitsch. *The Resolution Calculus*. EATCS Texts in Theoretical Computer Science. Springer, 1997.

[6] D. A. Plaisted. Complete problems in the first-order predicate calculus. *Journal of Computer and System Sciences*, 29 (1):8–35, 1984.

[7] R. A. Schmidt. *Optimised Modal Translation and Resolution*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1997.

[8] R. A. Schmidt. Decidability by resolution for propositional modal logics. *Journal of Automated Reasoning*, 22(4): 379–396, 1999.

[9] R. A. Schmidt. MSPASS, 1999. `http://www.cs.man.ac.uk/\textasciitilde{}schmidt/mspass/`.

A Small Linear Logic Planner

Alan Smaill*

*School of Informatics
University of Edinburgh
A.Smaill@ed.ac.uk

Abstract

Linear Logic programming languages allow resource tracking to be supported by the language itself; as such they give a way to implement planning systems that allow distributed and non-deterministic execution of the plans that are constructed.

Introduction

Various folk have suggested at different times that Girard's *linear logic* (3) might provide a good basis for reasoning about plans, given that this is a logic of *resources*. Samson Abramsky proposed a computational reading for intuitionistic linear logic (1), and I will use this along with Stephen Cresswell's work (2) as a starting point. In this framework, executable plans can be extracted from proofs in linear logic, these plans being suitable for distributed execution. Here I want to show how part of this can be expressed in a linear logic programming language with some higher-order features.

Linear logic programming

I use the linear logic programming language lolli (4) which implements an executable subset of the intuitionistic linear logic described in Cresswell (2); the aim is to let the linear logic machinery of lolli cope with resource management, and independence of resources, while attaching extract terms to inference steps in the logic.

In linear logic, a statement of the form $A \multimap B$ indicates that the resource B can be generated by consuming the resource A. Planning operators are regarded as reusable resources that allow $A \multimap B$ to be achieved, and the final plan uses some plan combinators together with the basic operators to hand. The Prolog-like syntax `H :- B` is equivalent in this setting to $B \multimap H$.

Resources are combined with the tensor operator, a form of conjunction: to establish a resource $A \otimes B$ means establishing A and B from disjoint sets of current resources; a given resource $A \otimes B$ makes available both resource A and resource B. In lolli, the comma represents the tensor operator (as it corresponds to conjunction in intuitionistic sequent calculus, where it only appears on the l.h.s. of sequents).

A mini planner

There are higher-order features in lolli that enable the easy construction of what in a first-order language is called a meta-interpreter; here statements can also serve as arguments to higher-order predicates. Unification is first-order (with an occurs check). The idea is to express the operators as procedures corresponding to transformations of the given resources.

The proving/planning engine should respect derivability in the logic, and attach extraction information while doing so.

For non-deterministic actions, another linear connective is used; a resource $A \oplus B$ supplies us with either A or B, where we have to work with whatever we are offered. This connective takes us out of the executable subset in lolli when use in a clause body. However, we can treat this soundly using the "with" connective $\&$, since $A \oplus B \multimap G$ is equivalent to $(A \multimap C) \& (B \multimap C)$.

Here is small implementation of a forward chaining planner using these ideas. It has a basic planning predicate, and an iterative deepening variant that will find smallest (most parallel) plans first.

The initial planning resources and the operators are given in separate modules. Plans are made on the basis of the (implicit) currently available resources. The logic guarantees that parallel actions do not interfere.

```
MODULE operators1.

% the operators

operator  (puton X Y)                  % the name
          (clear X)                    % pre-condition, not in delete list
          ( (clear Y, on X Z)          % resources consumed
             -o (clear Z, on X Y) ).   % resources generated

MODULE llplan3 operators initmodule.

id_plan P Goal :-  operators --o initmodule --o (idp  P Goal 1).

idp P Goal N  :- plan_bdd P Goal N.
idp P Goal N  :- M is N+1,
                 write_sans "searching at depth ", write M, nl,
                 idp P Goal M.

plan_bdd (op Op)  Goal _  :-              % goal by single op
        operator Op Pre (Now -o Next),
        Pre, Now,
        Pre -o Next -o (Goal, erase).

plan_bdd (par P1 P2) (G1,G2) N  :-         % par with depth bound
        plan_bdd P1 G1 N,
        plan_bdd P2 G2 N.

plan_bdd (then (op Op) Plan) Goal N  :-     % operator decreases depth bound
        N > 0, M is N-1,
        operator Op Pre (Now -o Next),
        Pre, Now,
        Pre -o Next -o (plan_bdd Plan Goal M).

plan_bdd (case( [A,Plan1], [B,Plan2])) Goal N  :-  % split on whether A,B holds
        N > 0, M is N-1,
        nd_operator Op Pre (Now -o (A; B)),
        Pre, Now,
          (Pre -o A -o (plan_bdd Plan1 Goal M))    % solve both branches
        & (Pre -o B -o (plan_bdd Plan2 Goal M)).   % with same resources
```

This is in fact a declarative lolli program, without cuts or negation by failure. It implements iterative deepening forward search when used with a Prolog-like depth first search engine as found in lolli.

References

[1] Abramsky, S. (1993), 'Computational interpretations of linear logic', *Journal of Theoretical Computer Science* **111**(3–57).

[2] Cresswell, S. (2001), Deductive Synthesis of Recursive Plans in Linear Logic, PhD thesis, University of Edinburgh.

[3] Girard, J.-Y. (1987), 'Linear logic', *Journal of Theoretical Computer Science* **50**, 1–102.

[4] Hodas, S. and Miller, D. (1994), 'Logic programming in a fragment of intuitionistic linear logic', *Information and Computation* **110**(2), 327–365.

Managing Mathematical Workflows

Volker Sorge[⋆]

[⋆]School of Computer Science,
University of Birmingham, UK,
V.Sorge@cs.bham.ac.uk

Over the last years there have been several projects on building environments to support the interconnection of mathematical software systems (2; 5) as well as some attempts to exploit these systems to support exploratory mathematics (1). However, all of these attempts are generally built for proving a single, already given goal. That is, the theorem that is to be shown, is already fully formulated or at least known to a degree such that only preconditions of the theorem's statement have to be adjusted. Our recent experience has shown, that systems geared towards proving single theorems are generally neither powerful nor flexible enough to deal with experimental mathematics. In particular, when assembling a final theorem by interleaving sequences of discovery, computation, and verification steps (3) various techniques have to be experimentally combined until a working sequence is found. The aim of our work is therefore to provide a framework, in which a working mathematician can easily automate a large number of experiments by specifying a mathematical workflow between a multitude of systems. While it is necessary to guarantee automatic interoperability between the systems in the framework, it should also allow the user to easily specify and implement additional transformations on data when it is passed between systems.

We motivate our ideas with two examples of mathematical workflows that originate from our work on constructing and verifying classification theorems in quasigroup theory. Figures 1 and 2 present two UML sequence diagrams for these examples. The first is concerned with the construction of classification theorems in finite algebra using a bootstrapping algorithm (for details see (3)). The main idea of the workflow is as follows: given a set of axioms we construct two algebras Q and Q' with a model generator, then shown them to be non-isomorphic by a theorem prover, construct a discriminant P with HR, and subsequently show that P (1) is a proper discriminant for isomorphism, (2) holds for Q, and (3) does not hold for Q', using again an ATP. Having these properties established, we can then either show that the original axioms together with P characterise an isomorphism class or construct another algebra that satisfies the axioms together with P, but is non-isomorphic to Q. Both problems are first simplified with a computer algebra system. The former problem is then tackled with a SAT solver and if successfully proved we stop. The latter problem is again passed to a model generator and if a model can be found, we restart the procedure. Analogously we can do the same for Q' and $\neg P$. The workflow eventually leads to a decision tree that represents a fully verified classification theorem.

The approach works well for classification with respect to isomorphism, it does not necessarily work for other equivalence relations. As a second example we take a look at the harder task of classifying loops (quasigroups with a unit element) with respect to isotopy. While for the overall construction of the classification theorem the same workflow as in Figure 1 can be used, the challenging part is to construct suitable isotopy discriminants since HR can no longer help. For this we have to establish another workflow, depicted in Figure 2, which is based on the concept of universal identities that form isotopy invariants (4). The workflow proceeds as follows: Given a generated equation E we check whether a loop of size ≥ 8 can be generated that satisfies E. If this is the case, E is transformed into a universal identity I. (In the workflow the symbolic computation unit essentially performs the transformations necessary in between the computations of the other two systems.) We then verify, with a first-order ATP, whether I is indeed an isotopy invariant. If it is not it is discarded, otherwise I can be used for discriminating between two concrete loop structures of order n during the classification. Before doing so, however, we first check whether I is at all useful by testing with a model generator that a loop of order n exists. Naturally, this does not guarantee that I can actually discriminate between the loops under consideration in the overall classification process. Therefore, we first systematically compute tens of thousands of equations and start the workflow with them. Then the model generator and the prover essentially have a filter function. This results in several thousand invariants which can be used during the classification. However, in case these are not enough, we can always restart the workflow in order to generate more.

References

[1] B. Buchberger, et. al. Theorema: An Integrated System for Computation and Deduction in Natural Style. *Proc. CADE-15 Workshop on Integr. of Deductive Systems*, p. 96–102, 1998.

[2] J. Calmet and K. Homann. Towards the Mathematical Software Bus. *Theoretical Computer Science*, 187(1–2):221–230, 1997.

[3] S. Colton, A. Meier, V. Sorge, and R. McCasland. Automatic generation of classification theorems for finite algebras. *Proc. of IJCAR 2004*, *LNAI* 3097, pages 400–414. Springer, 2004.

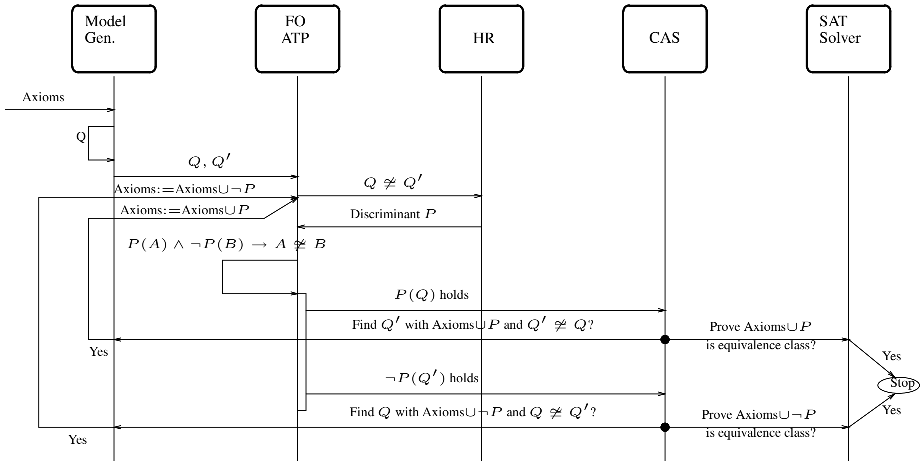

Figure 1: Workflow for constructing isomorphism classification theorems.

[4] E. Falconer. Isotopy invariants in quasigroups. *Transactions of the American Mathematical Society*, 151(2):511–526, October 1970.

[5] J. Zimmer and M. Kohlhase. System Description: The MathWeb Software Bus for Distributed Mathematical Reasoning. In *Proc. of CADE-18*, 2002.

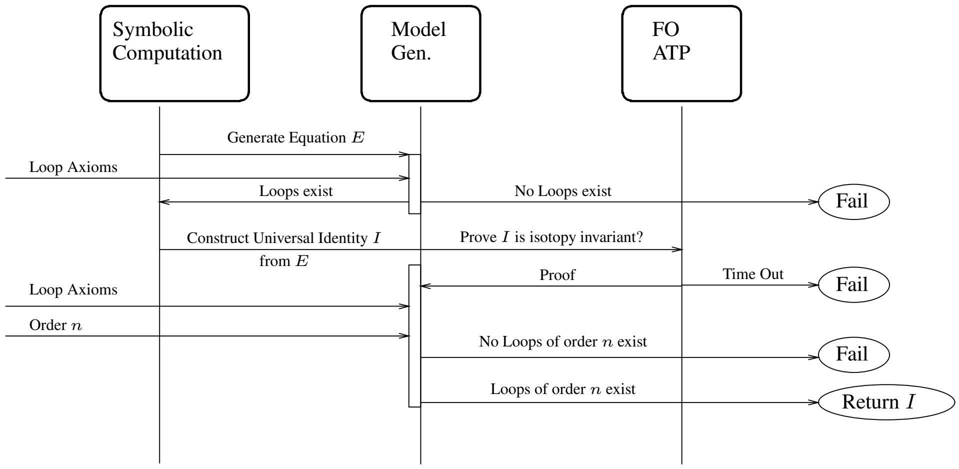

Figure 2: Workflow for constructing isotopy invariants for loops.

Using Model Generation in Automated Concept Formation

Pedro Torres[⋆] and Simon Colton[⋆]
[⋆]Department of Computing, Imperial College London, UK
180 Queen's Gate, Imperial College London, SW7 2AZ.
ptorres,sgc@doc.ic.ac.uk
First author supported by Fundação para a Ciência e a Tecnologia, grant SFRH/BD/12437/2003

Introduction

HR (1) is an Automated Theory Formation (ATF) system which makes extensive use of automated reasoning tools, in particular, the Otter theorem prover (3) and MACE model generator (4). Given a set of initial concepts, each provided with a definition and a set of examples, HR forms a theory about them. HR uses a set of production rules to form new concepts and forms conjectures according to the examples it finds for new concepts. For instance, if the set of examples for the new concept is contained in the set of examples of some old concept, an implication conjecture is generated. Similarly, equivalence and non-existence conjectures can be made. We present here a general first order setting for concept production in ATF, explain a recently implemented user-customisable generic production rule, and report on results in pure mathematics obtained using the new rule. The new concept production method make novel use of MACE, thus extending the combination of reasoning techniques employed in theory formation.

A general framework for concept production

General setting within first order logic Let $\{A_i\}_{i=1}^n$ be a collection of sets, which we will call *types*. Define $\mathcal{T}$ to be the set of all possible cartesian products made up of types, e.g. $A_1 \times A_2$, $A_2 \times A_2 \times A_5$, etc. We define a *concept* to be a subset of an element of $\mathcal{T}$, which means, equivalently, that a *concept* is an m-relation defined between any m types. We call m the *arity* of the concept. We write $\mathcal{C}$ for the set of all concepts. We will here consider concepts of the form $\{(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n | \ \phi(a_1, \ldots, a_n)\}$, where ϕ is some first-order logical formula for which the only free variables are $a_1, \ldots, a_n$ and any other variables are bounded. We will omit types whenever they are understood from context and write the above concept as $[a_1, \ldots, a_n] : \phi(a_1, \ldots, a_n)$.

A *production rule* is a function $\pi : \mathcal{D} \to \mathcal{C}$, where $\mathcal{D} \subset \mathcal{C}^q$, for some q, called the *arity* of π. The set of all production rules is denoted by Π. A production rule can be defined by choosing the transformation it operates on the logical formulas defining the old concepts. Suppose we have types A, B and C. We can define, for example, $\pi : (A \times B) \times (B \times C) \to A \times C$ as a function carrying out the mapping $([a, b] : \phi(a, b), [b, c] : \varphi(b, c)) \mapsto [a, c] : \exists_b.(\phi(a, b) \wedge \varphi(b, c))$. If we have examples for the old concepts, we can compute the examples of the new concept by applying the corresponding logical transformation. Starting with an initial pool of concepts $\{C_1, \ldots, C_k\}$, by recurringly applying production rules with appropriate domains, we can define new concepts, e.g. $C = \pi_3(\pi_2(C_1), \pi_1(C_2, C_3, C_4))$. The current version of HR has a set of 16 production rules, excluding the new rule introduced in this paper.

First order generic production rule In HR, all existing rules are fairly specific in the sense that, given a concept defined by some logical formula, ϕ, the logical transformation they operate always involves ϕ explicitly. In order to allow greater customisation by the user[1], the *first order generic production rule* was defined. This rule is a first step in defining, for algebraic domains, the general framework described above. It takes one concept of the form $[a_1, \ldots, a_n] : p(a_2, \ldots, a_n)$ and produces a new concept $[a_1] : q(a_2, \ldots, a_n)$ where the formula q can involve the formula p and is defined by the user. As an example, this rule can take the old concept $[a, b] : p(b)$ and produce the new concept $[a] : \exists_b.p(b)$. A more complex example could be to use the same old concept to produce the new concept $[a] : \forall_{b_1,b_2}.[p(b_1) \wedge p(b_2) \to (b_1 = b_2)]$. We can see the new production rule as a set of production rules, one for each first-order formula we choose. We will call each of these formulas a *parameterisation*. Before starting the theory formation process, a set of particular parameterisations needs to be defined by the user.

As can be seen from the examples above, parameterisations can take very distinct forms and we need a general way to compute the examples of the new concept produced by any parameterisation. To solve this, we coded HR to use MACE to compute the examples of the new concept. Suppose we start with some initial concept $[G, a_1, \ldots, a_n] : p(a_1, \ldots, a_n)$ and we wish to compute $[G] : q(a_1, \ldots, a_n)$. For each example of the initial concept, a particular groupoid G, we check if property q takes place. To do this, we supply MACE with: (i) the multiplication table of G, (ii) the definition of p and (iii) the definition of q involving p as per the parameterisation. We then use MACE in a model checking fashion to determine whether q holds for each G. The new concept produced will be formed by all groupoids G for which q holds.

[1]One conclusion from the study reported in (2) is that users of HR would appreciate the ability to define production rules.

Application The new production rule was used to explore algebraic structures, G, characterised by the single axiom $\forall_{a,b,c\in G}.(a*b)*c = b*(c*a)$. This is a special class of groupoids which are not necessarily associative or commutative and which we will call *star groupoids*. In order to study star groupoids, we defined two parameterisations for the new rule corresponding to the following logical transformations:

$$\begin{array}{lcl} [G,b]:p(b) & \mapsto & [G]:\exists_b.p(b) \\ [G,b]:p(b) & \mapsto & [G]:\forall_{a,b,c}.p(a)\wedge p(b)\wedge(c=a*b)\rightarrow p(c). \end{array}$$

Using HR's standard production rules in conjunction with the first order generic rule parameterised as above, we found some non-obvious results. Examples of star groupoids of sizes 1 to 7 where initially generated using MACE. The conjectures obtained were subsequently proved by Otter as part of the automated theory formation process. After a number of theory formation steps we were able to extract the following theorems, all resulting from the introduction of the new production rule:

$$\begin{array}{rl} (i) & \forall_b.(\exists_c(c*b=b)\leftrightarrow\exists_d(b*d=b)) \\ (ii) & \forall_b.(\exists_c(c*b=c)\leftrightarrow\exists_d(b*d=d)) \\ (iii) & \forall_{a,b,c}.(a*a=a\wedge b*b=b\wedge c=a*b\rightarrow c*c=c) \\ (iv) & \forall_{a,b,c}.(\phi(a)\wedge\phi(b)\wedge c=a*b\rightarrow\phi(c)) \end{array}$$

where $\phi(a)\leftrightarrow\exists_c.(c*a=a)$ which means "a has a left identity". These theorems can be respectively interpreted as: (i) every element of a star groupoid has a left identity iff it has a right identity; (ii) every element of a star groupoid is a right identity of some element iff it is a left identity for some (possibly different) element; (iii) the product of two idempotent elements is idempotent; (iv) the product of two elements which have a left identity, also has a left identity.

Concept production proof We are also interested in deciding whether a certain concept can be produced by a system of production rules. Suppose we have a set of initial concepts Γ and a set of target concepts Δ. Suppose additionally that a set of production rules Π is defined. In the spirit of theorem proving, we would like to implement a program that would take (Γ,Δ,Π) and output deductions of all target concepts Δ from the initial concepts Γ. A deduction of a target concept T is a particular composition of production rules that yields T starting only from concepts in Γ. These deductions can be seen as *concept proofs* stating that the target concepts are reachable by the production rule set and providing an explicit production path for them.

The general idea we considered to construct such a program is to view the production rules in the backwards direction, from a goal to subgoals. The goal will be the new concept a production rule generates and the subgoals are the old concepts needed for that production (production rules of arity 1 will only have one such concept and therefore only one subgoal). The initial goals will be the concepts we wish to produce and we would like to reach a stage where every subgoal is a concept in Γ. At each stage, the subgoals are refined by further backward application of the production rules until all the remaining subgoals are initial concepts. Using this *refinement* process, the proof tree is constructed from the root upwards. One of the key elements for solving the problems caused by the existence of variables and quantification is the process of determining the appropriate substitutions which will require *unification*. We believe that the problem of automating such a process may raise complexity issues. However, the typing information production rules carry with them may be able to reduce complexity considerably.

References

[1] S. Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag, 2002.

[2] S. Colton and S. Huczynska. The homer system. In *CADE-19*, 2003.

[3] W. McCune. The Otter user's guide. Technical Report ANL/90/9, Argonne National Laboratories, 1990.

[4] W. McCune. A David-Putnam program and its applications to finite first-order model search. Technical Report ANL/MCS-TM-194, Argonne National Laboratories, 1994.

Automated Reasoning for Dependently-Typed Languages

Matthew Walton[⋆]
[⋆]School of Computer Science and Information Technology
The University of Nottingham
mxw@cs.nott.ac.uk

Epigram(1) is a dependently-typed functional programming language currently under heavy development. A prototype implementation exists, but this is currently being replaced with Epigram 2, amidst a strong focus on extending the language's capabilities.

The dependent type system in Epigram burdens programs with a high level of proof. At type checking time they must show that the program's types are consistent; due to the complex nature of dependent types, this can be an extremely difficult task. At the present time, Epigram's inability to cope with some simple properties such as associativity makes many programs needlessly complicated, as the programmer must manually provide functions which give the type checker the information it needs to determine correctness.

Therefore, Epigram 2 will contain a mechanism for quoting relevant problems and passing them to an automated reasoning system, which can generate a proof and reflect that proof back to Epigram so that it can generate the appropriate code. This system must also be able to provide information about proof failure, as this information could be used to attempt a different proof strategy, or to provide useful reports to the programmer.

A motivational example is the problem of normal forms, specifically of monoidal structures. A monoid is a data type which has a value zero ($mzero$) and a binary operation plus ($mplus$). The zero acts as a zero to the plus operation, as 0 does to $+$ with the natural numbers, and the plus operation is associative. The natural numbers with addition can be easily seen to satisfy these laws and thus form a monoid. Another monoid frequently encountered and used in this example is the list, formed using $[]$ (the empty list) as the zero, and $(+\!\!+)$ (list append) as the plus.

While a monoid may be used to build structures in the form of trees using its zero and plus components, it is useful to be able to collapse these trees into a normal form, usually a list. Two trees of different structure may in normal form prove to be the same list, and doing the comparison on this level is the simplest way to eliminate any uncertainties caused by the associativity of the monoid plus and may also provide other benefits such as efficiency gains and code simplicity. Normal forms are of course applicable to many non-monoidal datatypes, but by considering only monoids we can reduce the problem to a reasonable size for an example.

In order to be able to use normal forms, Epigram needs to have proof that working with the normal form is ultimately equivalent to working with the original representation. We consider a type of expressions of natural numbers, which may be added. This is expressed in Haskell syntax thus:

$$\mathbf{data}\ Expr = Zero \mid Plus\ Expr\ Expr \mid Var\ Nat$$

This type forms a monoid with $Zero$ as the zero and $Plus$ as the plus. Var represents a variable with a known value. Note that Haskell does not possess a Nat datatype in its standard library to represent the natural numbers. Such a datatype may be easily imagined.

Expressions constructed with our type $Expr$ can be evaluated simply to natural numbers using a function $eval$, which is a *monoid homomorphism*: a function which converts one monoid into another, preserving structure. $eval$ converts $Zero$ to 0, the $Plus$ operation to $+$ and variables to their values. This preservation of structure may be proven using inductive methods.

We can express the normalisation of expressions to lists of natural numbers in a similar manner, mapping $Zero$ to the empty list, $Plus$ to $(+\!\!+)$ and variables to lists of one element. This is again a monoid homomorphism.

Now that we have a normal form, we need to be able to convert it to the natural numbers monoid, thus allowing us to produce the same end result from the normal form as we get with $eval$. Since our expressions consist solely of addition, in this case the appropriate operation is sum, which simply adds all elements of a list of numbers and returns the result. sum may be expressed as a monoid homomorphism, although the canonical definition in Haskell does not make this explicit.

with this knowledge of the preservation of the monoid structure across these three functions in place, all that remains is to convince Epigram that applying sum to the result of $norm$ is equivalent to the result of $eval$. This should not be difficult given the monoid laws and the knowledge that all three functions are monoid homomorphisms. Because each function preserves the monoid structure, the only possible loss of data is due to the associativity of the plus operator, which we need not be concerned about.

This pattern can be shown for any three monoids, and the proof at the abstract level of any three monoids is extremely clear, uncluttered by our existing knowledge about the algebraic properties of natural numbers. We therefore propose to implement a suitable inductive reasoning system which can produce proofs of this nature within Epigram, along with the necessary quoting and reflection mechanisms within Epigram which will allow it to operate on Epigram's internal representations of types.

Although the temptation is to simply write a reasoning system in Haskell and plug it into the Epigram code base, we also propose to investigate the possibility of allowing such reasoning systems to be written in Epigram itself. This could allow Epigram programmers in the future to supply their own reasoning techniques, allowing the system to cope with problems which the Epigram designers had not considered, or perhaps exploiting an efficiency gain to be found within the restricted set of types used by a given program.

References

[1] Conor McBride and James McKinna. The View From The Left. In *The Journal of Functional Programming, Vol.14(1), January 2004*.

Model Based Analysis of Industrial Scale Systems

Nikolay A. Zemtsov*

*Lomonosov Moscow State University
Leninskie gory 1, 119992, Moscow, Russia
nikolay.zemtsov@mtu-net.ru

Abstract

We are interested in static analysis of formal models that describe industrial scale systems. The goal of the analysis is, at first, to check partial correctness of the system and, at second, to construct finite state abstraction that would be used for monitoring the system behavior.

To specify formally industrial information systems we use the method of canonical process models developed by professor L.A.Kalinichenko and his group at the Institute of Problems of Informatics, RAS (see (2) for more references). This is an extensible model consisting of core and extensions of the core for every concrete process model. It is designed to serve as a common language, "Esperanto" for process models. The formal semantics was provided in Abstract Machine Notation (AMN) to enable use of B-method (1) and appropriate tools to formally analyze process specifications (descriptions of process behavior) and perform refinement proofs in a (semi-)automated way. Unfortunately, proofs in B-method require human interaction, what makes this approach difficult to apply in practice.

Another interesting question is how to search for process components using specification. Here also automated reasoning is applicable (ontological search, for instance).

Another direction of our research is monitoring and behavioral analysis of ASM (Abstract State Machines), see (4). This work also involves tasks directly related to automated reasoning. The main goal here is to develop efficient methods and tools for process analysis, under monitoring a few parameters of a system dynamics (3).

I am very thankful to Leonid Kalinichenko, Sergey Stupnikov and especially Rostislav Yavorsky for the very fruitful collaboration on the subject. I am very thankful to my wife Sveta and my daughter Anya for their support.

References

[1] J.-R. Abrial. The B-Book. – Cambridge University Press, 1996.

[2] L.A. Kalinichenko, S.A. Stupnikov, N.A. Zemtsov, Extensible canonical process model synthesis applying formal interpretation, Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems ADBIS'2005, Tallinn, Estonia, September 12 - 15, 2005

[3] Vladimir Filatov, Nikolay Zemtsov, and Rostislav Yavorskiy. Clusters of computations for a linear transition system. Proceedings of European Conference on Complex Systems. Paris, 14-18 November 2005. P. 84.

[4] Yuri Gurevich. Abstract State Machines: An Overview of the Project. In "Foundations of Information and Knowledge Systems", SLNCS, vol. 2942 (2004).