System Description: inka 5.0 - A Logic Voyager

Serge Autexier[1], Dieter Hutter[2], Heiko Mantel[2], and Axel Schairer[2]

[1] Saarland University, Fachbereich Informatik, Postfach 15 11 50,
D-66041 Saarbrücken, Germany, autexier@cs.uni-sb.de
[2] German Research Center for Artificial Intelligence, Stuhlsatzenhausweg 3,
D-66123 Saarbrücken, Germany, {hutter,mantel,schairer}@dfki.de

1 Introduction

Originally developed as an automatic inductive theorem prover [2] based on resolution and paramodulation, the inka system was redesigned in inka 4.0 in the early '90s [8] to meet the requirements arising from its designated use in formal methods. Meanwhile several large industrial applications of the verification support environment (VSE) [7] have been performed which gave rise to thousands of proof obligations to be tackled by its underlying deductive system inka.

The new version of inka is a result of this long experience made in formal software development. Thus, the major improvements of inka 5.0 are concerned with the requirements arising when dealing with large applications. The user database is distributed along different deductive units each of which consists of an individual logic (consequence relation) and a set of (local) axioms. In order to allow for the logical implementation of structured specifications as they are provided in languages like CASL [3], the deductive units may import the deductive reasoning of other units with the help of morphisms. Relationships between different units are also postulated with the help of morphisms between two units which give rise to various proof obligations. inka also supports the evolutionary aspect of formal software development as it incorporates a management of change. It minimizes the proof obligations arising when changing a deductive unit or defined relationships between some units. As a basis for the implementation of different logics, inka provides an annotated λ-calculus as an underlying meta-language. Annotations are a generalization of the colour concept [6] and are used to incorporate domain knowledge into the proof search process. inka provides a uniform hierarchical proof datastructure and a generic tactic definition mechanism to implement appropriate proof search engines.

2 System Description

Meta-level Language. inka supports the use of formulas of different logics and proof-objects of different calculi. All these objects are encoded in a meta-level language, based on a (ML-type) polymorphic λ-calculus [4]. λ-terms are automatically kept in $\beta\eta$ long normal-form. Using λ-calculus as meta-language provides a clear concept of variables, namely bound and free variables.

16th International Conference on Automated Deduction, CADE -16, Trento, Italy, LNAI 1632, pp. 207–211, Springer, July 1999.

The annotations in inka's λ-calculus are described in a first-order term language and are attached to occurrences of function constants and variables, to λ-abstractions and applications. The concept of annotations is a generalization of the concept of colours [6], which has been developed in the context of inductive theorem proving to encode the knowledge about the similarities between induction hypothesis and induction conclusion [9]. They are used by tactics to encode domain knowledge into logical objects and the underlying annotated calculus supports the inheritance of the annotations. For a detailed description of annotated λ-calculus see [10,6].

The implementation of the annotated λ-calculus also comprises a unification for annotated λ-terms with free variables [10,6]. The unification algorithm is generic in the sense, that domain specific unification algorithms can be integrated into it. Thus, specialized unification algorithms, which make use of semantic properties of defined function constants, can be implemented and are integrated in an object-oriented manner into the generic unification algorithm.

Hierarchy of Logics. inka provides a mechanism for user defined logics in terms of so-called logic units (cf. Figure 1). The rôle of a logic unit is to define a truth-type and a language of logical formulas. In addition to the declarative content, a logic unit may contain specific domain-knowledge. For example, a logic unit can contain a specific theory unification algorithm for formulas; for first-order logic, a specific unification algorithm is implemented, which moves quantifiers over other connectives. Each time a logic unit is used, the specific unification is linked into the generic unification from the underlying λ-calculus. This approach differs essentially from logical frameworks like LF [5] or Isabelle [12] in which logics, respectively, are represented as signatures in a dependent type theory or by an embedding into a meta-logic.

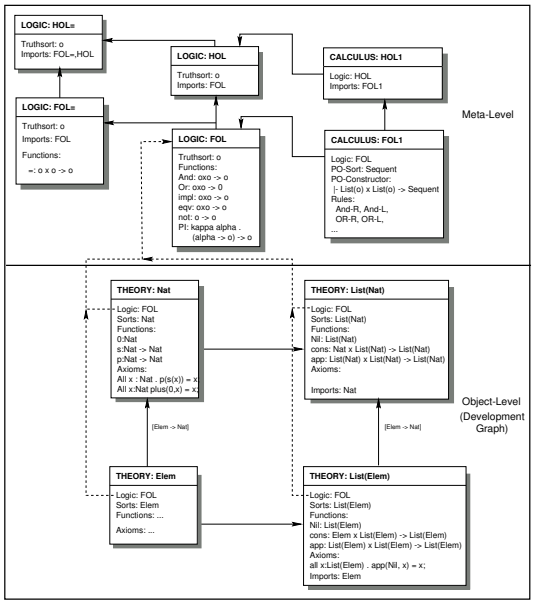

Fig. 1. The hierarchy of logics and the development graph

In inka, a calculus is represented within a calculus unit which is related to a logic unit and defines the proof objects as well as the basic calculus rules. E.g., in the sequent calculus unit for first-order logic, the proof-objects are sequents, which are build from lists of first-order logic formulas. The implementation of the basic calculus rules as well as the tactics for proof search are attached to the

calculus units. If a calculus unit is used, its tactics are linked into the generic deduction mechanism which supports forward as well as backward application of rules. inka allows for an implementation of different calculi for different logics. The tactics and calculus rules are implemented using the generic tactic definition mechanism described in the paragraph Deduction. The proofs are constructed by a uniform tactic mechanism and are represented in an uniform proof datastructure (cf. Paragraph Deduction).

Furthermore, different kinds of relationships between logics are explicitly represented by logic morphisms. Along these morphisms, formulas of one logic can be transformed into equivalent formulas of another logic.

Development Graph. It has long been recognised that so-called *specifications in the large* are only manageable if they are built in a structured way on the basis of smaller specifications. Specification languages, like for instance CASL [3], provide various mechanisms to combine basic specifications to extensive structured specifications.

inka supports the use of such structured specifications and thus also the structured deduction by distributing the resulting logical axiomatization into so-called deductive units. Each deductive unit describes a logical theory and corresponds to a basic specification as defined in [3]. A deductive unit is linked to a calculus unit defining its underlying consequence relation. The units can be connected via consequence morphisms allowing the user to import (mapped) theories to a deductive unit. The same mechanism can be used to postulate relations between deductive units. Drawing a so-called theorem link from a unit N_1 to a unit N_2 (wrt. a morphism σ) gives rise to the proof obligation that the mapped theorems of N_1 are valid within N_2. inka provides several techniques to minimize the arising proof obligations by making use of already existing relations between other units linked to N_1 and N_2.

Deduction. The application of formal methods in an industrial setting (cf. [7]) results in an increased complexity of the specification and the correspondent verification. Tackling arising proof obligations from industrial case studies, the proofs are too complex to be done fully automatically but they are also too longish to be done by hand. Thus there is a need to combine a high degree of automation with an elaborate user interface to advise the deduction system in case where built-in strategies are too weak to find the proof automatically.

All deductions in the inka-system are explicitely represented in a uniform hierarchical proof datastructure, which is a generalization of the hierarchical proof datastructure (PDS) developed for the ΩMEGA-system [1]. It is realized by an acyclic directed graph, whose nodes are annotated by proof-objects defined by some calculus and whose edges are annotated by basic calculus rules or tactics. The lowest proof level consists of edges annotated only by basic calculus rules which are related to higher edges annotated by tactics. This datastructure allows to view a proof on different levels of abstractions. Hence, the proof can be communicated to a user on different levels of abstraction, which is an adequate mechanism to support interactive proof construction.

Tactics and calculus rules are implemented by a generic tactic definition mechanism, which allows for a simple implementation of tactics and rules and hides necessary updates of the proof graph from the tactic engineer. The tactic definition mechanism allows for the tactic engineer to focus on parts of a proof-object. Foci are explicitely represented and the designed tactics can manipulate the content of their argument foci, without changing the foci themself. This is a simple and efficient mechanism, offering the necessary level of abstraction to a tactic engineer to support tactic design. The tactic definition mechanism already existed in the old inka-system [8] and has been adapted to the hierarchical proof representation and in order to allow for the manipulation of foci wrt. different proof-objects within one tactic.

Interfaces. The user interface of the inka-system is an adaptation of the interface system $\mathcal{LOUI}$ [13], which is a user interface for the ΩMEGA-system [1]. $\mathcal{LOUI}$ has been adapted to be a generic user interface for theorem provers as for instance a visualization of the development graph has been added. It is implemented in the distributed Oz programming language [11] and the communication with inka is done via a socket-communication. The interface between Lisp and Oz is an abstract representation of the datastructures to be visualized while the actual layout is done in Oz. This minimizes the communication effort between Lisp and Oz and results in a sufficient speed of the visualization process.

The inka specification language is designed to include a subset of the *Common Algebraic Specification Language (CASL)* [3], which is currently developed to define a standard language for algebraic specifications.

3 Progress, Availability and Future

The inka 5.0 system is an experiment in providing a generic software verification system. It is still in an incomplete, prototypical state. However, all basic mechanisms described in this paper are implemented and used for some example logics and sequent calculus proof search.

The core inka is implemented in Allegro Common Lisp. The interface runs on distributed Oz, which is available for Unix and Windows.

As a next step we intend to integrate a logic for algorithmic function and predicate definitions as well as the methods to prove their termination as tactics. Termination proofs can be inspected and already proven lemmata can be used during the construction of termination proofs, which are the main advantages wrt. the black box implementation of these methods in the old inka system [8].

References

[1] C. Benzmüller et al.: ΩMEGA: Towards a mathematical assistant, In W. McCune (ed), *CADE-14*, Springer, LNAI 1249, 1997.

[2] S. Biundo, B. Hummel, D. Hutter, C. Walther: The Karlsruhe Induction Theorem Proving System. In Jörg H. Siekmann (ed), *CADE-8*, Springer, LNCS 230, 1986.

[3] CoFi-task group on language design, ESPRIT working group 29432, EU, 1998. CoFI Webpage: http://www.brics.dk/Projects/CoFI/

[4] L. Damas, R. Milner: Principal type schemes for functional programs, *Ann. ACM Symp. on Principles of Programming Languages (POPL)*, 1982.

[5] R. Harper, F. Honsell, G. Plotkin: A framework for defining logics, *Journal of the Association for Computing Machinery*, 40(1), 1993.

[6] D. Hutter, M. Kohlhase: Managing Structural Information by Higher-Order Colored Unification, *Journal of Automated Reasoning*, accepted, 1999.

[7] D. Hutter et al.: Verification Support Environment (VSE), *Journal of High Integrity Systems*, Vol. 1, 1996.

[8] D. Hutter, C. Sengler: inka - The Next Generation. In M. McRobbie, J. Slaney (ed), *CADE-13*, Springer, LNAI 1104, 1996.

[9] D. Hutter: Guiding Induction Proofs, In M. Stickel (ed), *CADE-10*, Springer, LNAI 449, 1991.

[10] D. Hutter: Coloring terms to control equational reasoning, *Journal of Automated Reasoning*, Vol. 18, 1997.

[11] Programming System Lab, Saarland University, Saarbrücken, 1998. The Oz Webpage: http://www.ps.uni-sb.de/ns3/oz/

[12] L.C. Paulson: Isabelle, A Generic Theorem Prover, Springer, LNCS 828, 1994.

[13] J. Siekmann et al.: A Distributed Graphical User Interface for the Interactive Proof System ΩMEGA. In R. C. Backhouse (ed), *UITP98*, Eindhoven Technical University, Report 98-08, 1998.