# UniForM Perspectives for Formal Methods

Bernd Krieg-Brückner

Bremen Institute of Safe Systems
TZI, FB3, University of Bremen
P.O. Box 330440, D-28334 Bremen
bkb@Informatik.Uni-Bremen.DE

**Abstract.** Trends for Formal Methods are reviewed and illustrated by several industrial applications: logical foundations of combination, verification, transformation, testing, and tool support. The UniForM Workbench is the background for highlighting experiences made over the past 20 years.

## 1 Introduction

This paper outlines some state of the art technology in Formal Methods and attempts to extrapolate towards the next century. The UniForM Workbench (Universal Formal Methods Workbench, cf. Krieg-Brückner et al. 1996) is developed by the Universities of Bremen and Oldenburg, and *Elpro*, Berlin, funded by the German Ministry for Education and Research, BMBF. In its present state, it provides a focal point to

- review experiences in the past decades developing languages, methods and tools,
- give an illustrative example of state of the art technology,
- evaluate preliminary experiences with industrial applications, and
- assess the industrial potential for the future and illuminate technological trends.

## 2 Logical Foundations

The first use of Formal Methods in software development, regarded by many as the most prominent, is for modelling, using a mathematically well-founded specification language. The need for modelling arises in many aspects and properties of software, or more generally systems: for the physical environment of a hybrid hardware / software system, for the timing behaviour and real-time constraints of an embedded system, for the hazards and safety requirements of a safety-critical system, for the concurrent interactions of a reactive system, for deadlock and livelock prevention, for performance and dependability analysis, for architectural and resource requirements, and, finally, at many stages of the software development process for requirements and design specifications, etc., to the implementation of a single module. The important distinction between property-oriented and model-oriented specifications is now universally accepted: the former are sufficiently loose to avoid over-specification, cap-

D. Hutter, W. Stephan, P. Traverso, M. Ullmann (eds.):
*Applied Formal Methods – FM-Trends 98.*
*International Workshop on Current Trends in Applied Formal Methods.*
Lecture Notes in Computer Science 1641. Springer (1999) 251-265

turing just the necessary properties and leaving freedom of choice for the implementor; the latter are geared towards describing a particular, operational implementation, giving sufficient detail to allow e.g. detailed performance analysis.

The second and equally important use of Formal Methods is to provide support for correct development, i.e. mathematical notions of refinement or implementation that guarantee correctness preservation w.r.t. the initial requirements, be it by the invent-and-verify paradigm, synthesis or transformation.

### ~~1.1~~2.1 Combination of Formal Methods

How can we ever hope for a unique standard formalism to cover all the needs listed above? Instead, the solution is a variety of formalisms that complement each other, each adapted to the task at hand: specification languages and development methodologies, specific development methods or proof techniques, with a whole spectrum of tool support. Thus the challenge is to cater for correct combination of formalisms to

1. ensure correct transition from abstract to concrete specifications when switching between formalisms during the development process ("vertical composition"),
2. ensure correct combination of formalisms in a heterogeneous situation, e.g. combining concurrent and sequential fragments ("horizontal composition"),
3. attach various kinds of tools, e.g. for performance or deadlock analysis, and
4. provide "hooks" for specifications that do not affect semantics but allow the use of tools, e.g. ordering of operation symbols or equations for effective rewriting to prototype, or guidance for design choices w.r.t. expenditure of resources such as speed vs. space, or relative usage of operations of an abstract data type.

Such combinations are by no means easy to achieve. The need for research on the first two has been recognised and requires demanding mathematical foundations, such as advanced methods in category theory. This has lead to languages for "institution independent" heterogeneous composition of modules ("in the large", see e.g. Astesiano and Cerioli 1994, Tarlecki 1996, Diaconescu 1998); approaches for reasoning about correct composition of the logics capturing the semantics "in the small" (see e.g. Mossakowski 1996, Mossakowski et al. 1997, 1998b, Sernadas et al. 1998a, 1998b) introduce notions such as *embedding, translating* one formalism to another (cf. (1) above), *combination* of two formalisms, or *projecting* to either from the combination.

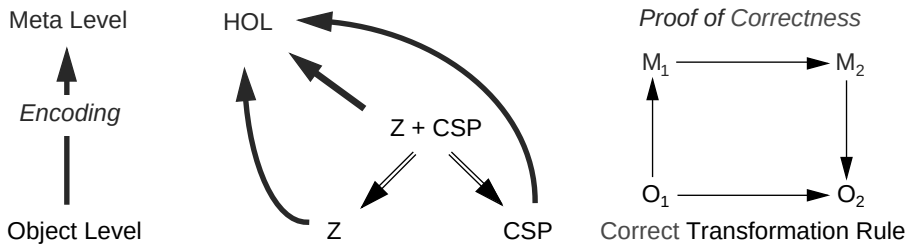


**Fig. 1.** Semantic Representation in UniForM

**Semantic Representation.** The approach of UniForM is to represent the semantics underlying a particular formalism or language in higher-order logic (HOL) as it is realised in the logical framework Isabelle (Paulson 1995). Fig. 1 shows a tiny Logic Graph for Z, CSP and their projections from the combination Z+CSP, plus the logic encoding into HOL at the meta level. Specifications in these languages are represented as theories in Isabelle and used for theorem proving with the verification system IsaWin on top of Isabelle (cf. section 3.1), and, as a basis for transformational development (cf. section 3.3), for proving the correctness of transformation rules.

**HOL-Z, HOL-CSP and HOL-CASL.** In HOL-Z, the logic of Z has been represented (cf. Kolyang et al. 1996a, 1996b, 1997, Kolyang 1997, Lüth et al. 1998b) and the mathematical tool kit has been proved correct (in co-operation with the ESPRESS project); this resulted in ca. 1k theorems, a 4k line proof script, and ca. 3 person-years of effort.

HOL-CSP represents the logic of CSP; a small but pervasive error in the 20 year old theory of CSP has been found and corrected (cf. Tej and Wolff 1997, Tej 1999). The process algebra has been proved correct; this resulted in ca. 3k theorems, a 17k line proof script, and ca. 3 person-years of effort. The example shows that such an endeavour is by no means trivial but pays off in the end. The proof of correctness of transformation rules, in particular, is now much easier.

The above statistics includes the effort of becoming familiar with the intricacies of Isabelle, and most of the effort went into the proof of the process algebra of CSP. A subsequent representation of the logics and static semantics of CASL basic specifications (including an intricate overloading resolution) only required about 1 person-year of effort (see Mossakowski et al. 1998a).

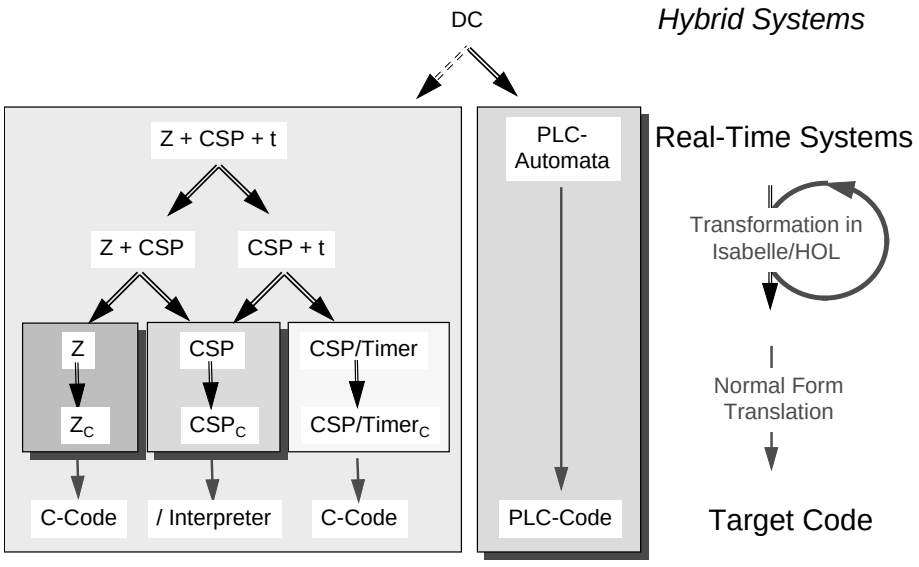


**Fig. 2.** Method Combination in UniForM

**Reactive Real-Time Systems.** The first instantiation of UniForM has been for Z and CSP since these are considered to be rather mature and relatively well established in industry. At the moment, we are working on methods ("structural transformations") to project not only from Z+CSP (actually Object-Z, cf. Fischer 1997), but also from CSP+t, i.e. CSP with real-time constraints, to CSP without such constraints on the one hand, and simple timer processes on the other, cf. fig. 2. Thus specialised methods can be used in the projected domains. This breakdown is also successfully used for testing of real-time and hybrid systems (cf. section 3.1).

A special class of hybrid systems, so-called PLC-Automata, generate real-time programs for Programmable Logic Controllers directly (cf. Dierks 1997, Tapken 1997, 1998). They have a semantics as DC-implementables, a subset of Duration Calculus (Zhou et al. 1992). Further extensions towards hybrid systems are planned.

### ~~1.~~2.2 Standard Family of Specification Languages

A standard formalism for all aspects of formal methods seems pragmatically undesirable (if not impossible) since a projection to a restricted and supposedly simpler formalism allows easier reasoning and specialised tools. However, standardisation should be aimed for in well-defined areas. IFIP WG 1.3 (Foundations of System Specification), based on more than 7 years of experience of the ESPRIT WG COMPASS, (cf. e.g. Krieg-Brückner 1996), started the Common Framework Initiative for Algebraic Specification and Development, CoFI.

CoFI, an international effort by primarily European groups, is developing a family of specification languages, a methodology guide and associated tools (see CoFI, Mosses 1997). The major language in this family, the Common Algebraic Specification Language CASL, has just been completed; it is the basis for sublanguages and extensions in the family. Its formal semantics only awaits a final revision. Various parsers exist as well as a prototype implementation of the static semantic analysis for basic specifications in Isabelle for the UniForM Workbench (Mossakowski et al. 1997); it allows theorem proving and will be the basis for transformational development.

CASL is a rather powerful and general specification language for first-order logic specifications with partial and total functions, predicates, subsorting, and generalized overloading (cf. CoFI, Cerioli et al. 1997). Sublanguages of CASL, in connection with the planned extensions towards higher-order, object-oriented and concurrent aspects, allow interfacing to specialised tools and mapping from/to other specification languages (cf. Mossakowski 1998); this aspect is crucial for its intended impact.

## 3 Verification and Validation

Formal Methods are meant for the development of dependable systems: apart from safety and security, aspects of availability, reliability, fault-tolerance, and a general adherence to functionality requirements are important. Thus correctness is only one aspect, but obviously at the heart of the matter. In particular in safety-critical do-

mains, application developers become increasingly aware of the importance of methods *guaranteeing* correctness w.r.t. a formal specification, the "contract".

### ~~1.1~~3.1 Testing vs. Correctness Proofs

In many applications, complete formal development is regarded as unrealistic so far; the technology is only emerging. Often, only the critical kernel is formally developed But even with large systems, formal methods become increasingly important:

**Verification, validation and testing.** Peleska (1996) has developed a methodology and the tool kit VVT (cf. also Peleska and Siegel 1996, 1997), presently being integrated into the UniForM Workbench, that allows *automatic* testing. Test cases are generated from a real-time specification; they drive the completed hardware/software system as a "black box" in a hardware-in-the-loop configuration from a separate computer containing the test drivers, simulating a normal or faulty environment. The testing theory ensures, that each test will make an actual contribution, approximating and converging to a complete verification.

Even more important in practice is that thousands of lines of generated test cases can also be checked automatically against the formal specification; manual inspection is quite impossible. This approach is very cost-effective. It has been applied successfully to part of the case study of UniForM, a control computer on board of a train for railway control, developed by *Elpro*, Berlin (cf. Amthor and Dick 1997), and to an electric power control component of a satellite developed by OHB, Bremen.

**Abstraction to verify special properties.** Another team lead by Peleska (Buth et al. 1997, 1998, Urban et al. 1998) has developed a technique for abstracting from an existing program to verify the absence of deadlocks and livelocks. It was applied successfully to more than 100k lines of Occam implementing a safety layer of a fault tolerant computer to be used in the International Space Station Alpha developed by DASA RI, Bremen; thus it is scalable and applicable to realistic applications.

The concrete program is abstracted to a formal specification in CSP containing only the *essential communication behaviour*; the approach guarantees, that the proof for the abstract program implies the proved property for the concrete one. If the proof fails, the property does not hold, or the abstraction is not yet fine enough. The task is split into manageable subtasks by modularisation according to the process structure, and a set of generic composition theories developed for the application. The modules are then model-checked using the tool FDR (Formal Systems Ltd., Oxford).

The abstraction was done by hand; future research will focus on implementing formal abstraction transformations in the UniForM Workbench to support the process.

### ~~1.2~~3.2 Model-Checking vs. Interactive Proofs

**Model-checking** is a very important technique in practice. The FDR tool is very useful for CSP, mostly for validating specifications, proving properties such as deadlock-freeness, and for development, proving the correctness of a refinement in the invent-

and-verify paradigm. But it can do more: the transition graph it generates can be interpreted at run-time; this technique has been used for the safety layer of a computer on-board a train (see above). The abstraction and modularisation method applied to the International Space Station, described in the preceding paragraphs, shows two things:

- Model-checking is extremely useful when the resp. data-types are essentially enumeration types and the systems small enough.
- For large systems, these properties are likely to be violated; reasoning about modularisation and composition properties is necessary; proof tools are desirable.

Thus both model-checking and (interactive) proofs should go hand in hand. In the UniForM Workbench, a link from the interactive proof tool (see next paragraph) to the FDR tool is presently being implemented.

Moreover, the experience of Haxthausen and Peleska (1998) when solving the train control problem in general has been, that reasoning about algebraic properties at a high level of abstraction is necessary, with subsequent refinements; model-oriented specifications and model-checking are not enough for this very practical problem that had defied a general solution thus far.

**Fig. 3.** The Isabelle Proof Assistant IsaWin in UniForM

**A Window to Isabelle.** The UniForM Workbench makes extensive use of the generic theorem prover Isabelle (Paulson 1995), and heavily relies on the possibilities for interaction and tactic definition. A graphical user interface, a "window to Isabelle", IsaWin, has been constructed that hides unnecessary details from the uninitiated user (see Kolyang et al. 1997, Lüth and Wolff 1998). Objects such as theories, substitutions, proof rules, simplification sets, theorems and proofs are typed (cf. fig. 3); icons

can be dragged onto each other or onto the manipulation window to achieve various effects. This graphical and gesture-oriented approach is as a major advance over the rather cryptic textual interface. In the example, a set of rewrite rules for simplification is dragged onto the ongoing proof goal in the manipulation.

**Architecture of the UniForM Transformation and Verification System.** In fact, theorem proving and transformation, both a form of deduction, are so analogous, that the UniForM Verification System IsaWin shares a substantial part of its implementation with the Transformation System TAS (cf. fig. 4, see Lüth and Wolff 1998, Lüth et al. 1998a). Like Isabelle, it is implemented in Standard ML; sml_tk (Lüth et al. 1996) is a typed interface in SML to Tcl/Tk; on top, the generic user interface GUI provides the appearance of fig. 3 and fig. 5. This basic system is then parametrized (as a functor in SML terminology) either by the facilities for theorem proving of IsaWin or those for transformation of TAS. In addition, both share focussing and manipulation of scripts, i.e. proofs or development histories.

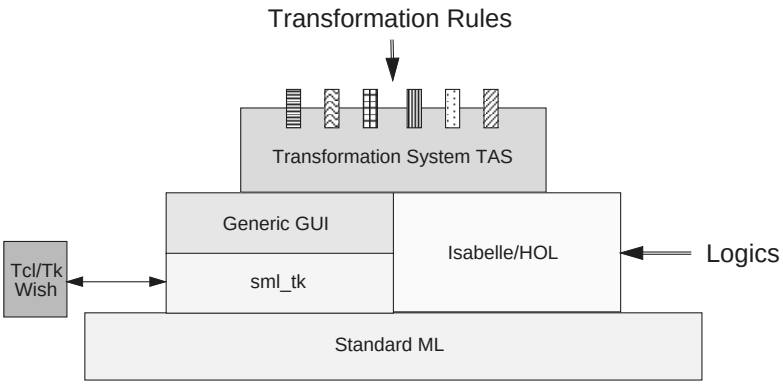


**Fig. 4.** Architecture of TAS, the UniForM Transformation System

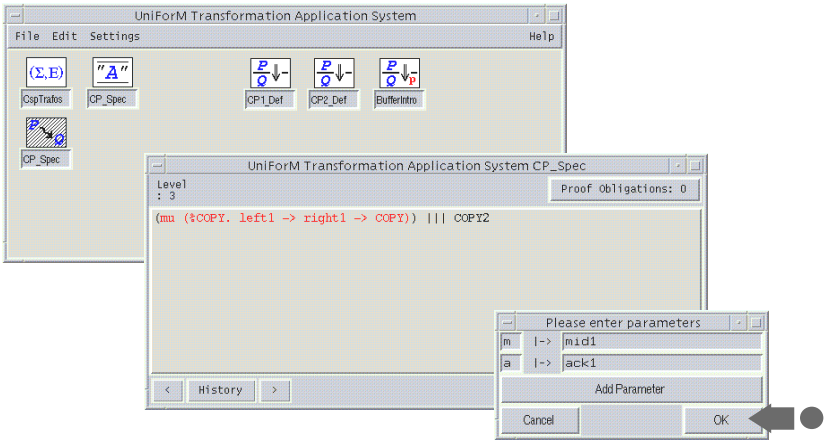


**Fig. 5.** Application of a Parametrized Transformation Rule

### ~~1.~~3~~3~~.3 Transformation vs. Invent-and-Verify

**Synthesis by Transformation.** While the invent-and-verify paradigm is already supported by IsaWin, we definitely prefer synthesis-by-transformation over invent-and-verify as the pragmatically more powerful paradigm. First of all, the latter can be implemented by the former as a transformation rule that generates the necessary verification condition from the applicability condition. Secondly, this *automatic* generation of the required verification conditions is precisely one of the advantages of the transformational approach. The developer can concentrate on the development steps (viz. applications of transformation rules) first while the verification conditions are generated on the side and tabled for later treatment. Above all perhaps, single transformation rules and automated transformation methods embody development knowledge in a compact and accessible form like design patterns. Transformation rules preserve correctness; they can themselves be proved correct in UniForM against the semantics of the object language, e.g. at the level of the logic representation in HOL, cf. fig. 1.

**TAS, the UniForM Transformation System.** TAS may be parametrized by logics (i.e. semantic representation of the formalism involved) at the Isabelle level, and by transformation rules at the level of TAS itself, cf. fig. 4 (Lüth 1997). On top of the basic architecture that it shares with IsaWin, TAS provides icons for (program or specification) texts, transformation rules, possibly parametrized, and transformational developments in progress, in analogy to proofs (cf. shaded icon and manipulation window in fig. 5). In the example, a parametrized transformation rule is applied to the highlighted fragment denoted by focussing, and a window for the editing of parameters is opened. Once input of parameters is completed, the rule is applied, and a further proof obligation is possibly generated. A proof obligation may be discharged during or after the development by transferring it to IsaWin or another verification system such as a model checker (presently FDR).

The functionality of TAS subsumes that of a forerunner, the PROSPECTRA system (cf. Hoffmann and Krieg-Brückner 1993). However, the basis of Isabelle allows a more compact, more flexible and more powerful realisation: parametrisation by additional transformation rules is a matter of minutes (instantiation of a functor rather than recompilation of the whole system!); static semantic analysis can often be mapped to type checking of Isabelle; proof tactics can be defined as SML programs and often allow the automation of applicability conditions, such that much fewer residual verification conditions need to be interactively proved by the user.

**Development History.** Note also the History button that allows navigation in the development history, in particular partial undo for continuation in a different way. The whole development is documented automatically and can be inspected in a WWW browser, see fig. 6. The example shows the development of a communication protocol with send and receive buffers by a sequence of transformations in CSP.

**Reusability of Developments.** The development history is a formal object as well, (partial) replay is possible. A development can be turned into a new transformation rule by command; the generated verification conditions are then combined to a new

applicability condition. Combined with abstraction, *developments themselves* become reusable in new situations, not just their products, i.e. modules (cf. Lüth et al. 1999).

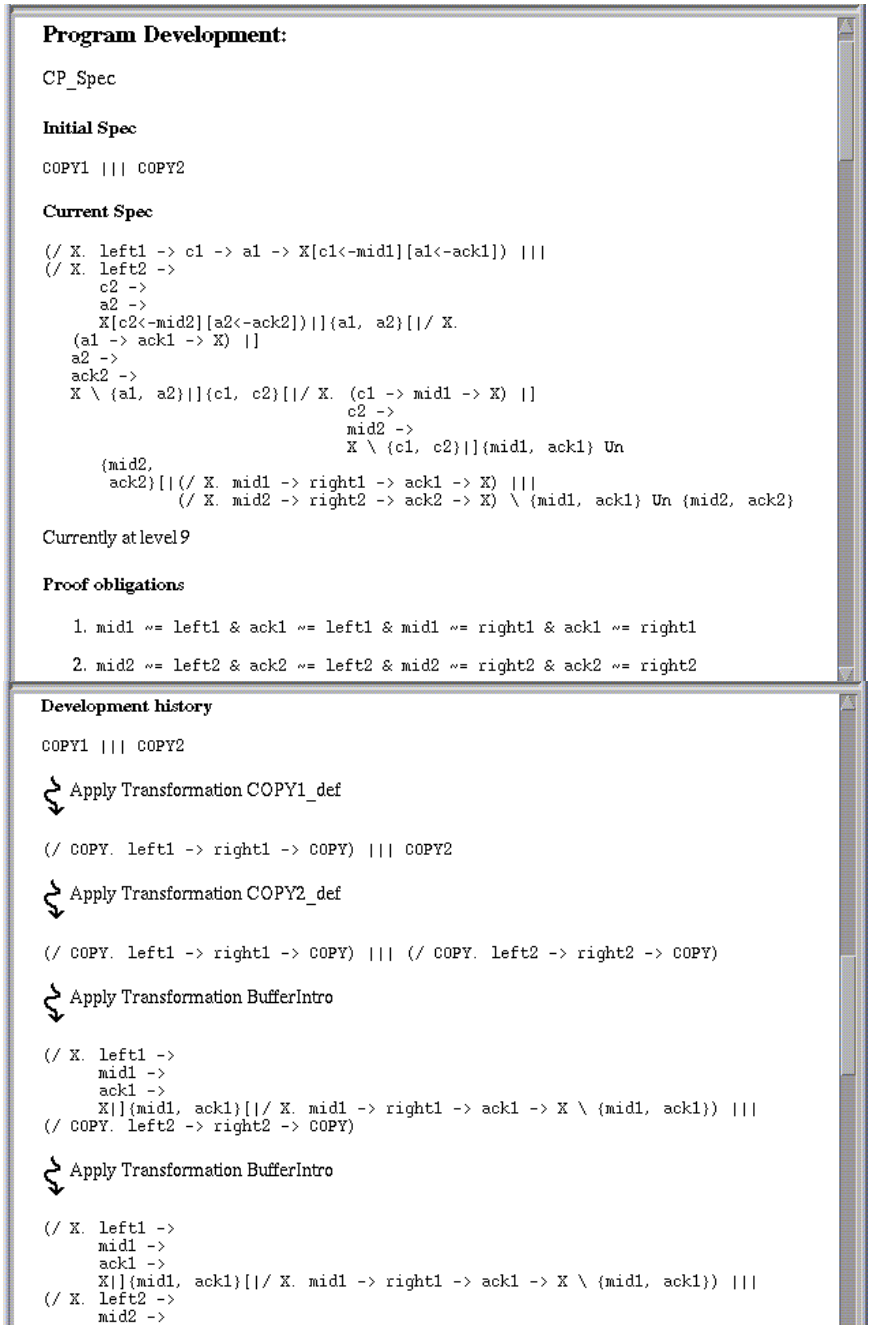


**Fig. 6.** Initial and Current Specification, Proof Obligations, and Development History

## 4 Integration into the Software Life Cycle

**Integration of Formal Methods into Existing Process Models** is important for success in industry. The Software Life Cycle Process Model V-Model (1997), originally a German development standard, has become internationally recognised. As many such standards, it loads a heavy burden on the developer by prescribing a multitude of documents to be produced. Thus tool support is essential to

1. tailor the V-model first to the needs of a particular enterprise, then
2. tailor the V-model to the special project at hand, fixing methods and tools,
3. support its enactment guiding and controlling the use of methods and tools, and
4. provide automatically generated development documents.

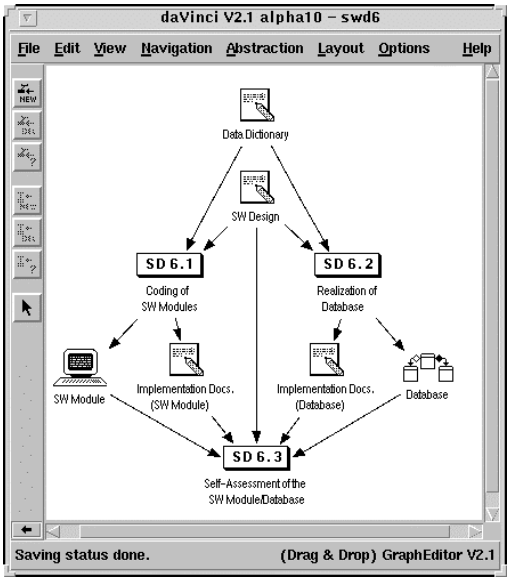


**Fig. 7.** Example of a V-Model Process Graph as supported by the UniForM Workbench

Blank Purper and Westmeier (1998) are presently developing the Graphical Development Process Assistant, adapting the V-model to formal methods, where development and quality assurance are intimately related. The V-model is presented as a heavily interwoven hypertext document, generated from a common database, and tools support items 1 to 4 above; cf. also fig. 7. Integration into a development environment such as the UniForM Workbench allows the coordination with its methods and tools (item 3). Tools themselves, on the other hand, can generate development documents in conformance with the V-model (cf. item 4), such as the development history of fig. 6.

**Isolation of Dependable Parts** is an issue that is at the heart of the combination problem; at the same time it offers possibilities for a kind of "divide and conquer" ap-

proach since methods and tools can be tailored to more specialised problems, cf. the projection into fragments with and without time for real-time systems in section 2.1.

**Combination of Conventional, Semi-Formal and Formal Techniques** arises naturally when interfacing with other methods in the context of the V-model. Safety considerations, and thus the employment of formal methods, will often be restricted to parts of a system. Ideally, graphical interfaces will give the illusion of working with an informal method while an underlying formal semantics provides hooks to the use of formal methods (cf. the PLC-Automata in section 2.1); this area has great potential.

At the same time, it is sometimes advisable to flip back and forth between informal techniques at a high level of abstraction, e.g. requirements analysis, and formal methods, once more detail is required; complete formalisation might be premature and rather a burden, but formal methods are already useful at an early stage to support the analysis. An example is the specialisation of fault trees for hazard analysis to develop safety requirements and safety mechanisms, cf. (Lankenau et al. 1998).

## 5 Tool Support

**Large, Integrated Systems** become unmaintainable dinosaurs very fast, in particular when several distant development teams are trying to coordinate, and maintenance threatens to cease.

**Collections of Loosely Coupled Specialists** are the obvious solution – unfortunately often *large* specialised tools, given the complexity of formal methods. It is most important that each tool can be developed and maintained by one or very few persons such that an overview over its functionality and integrity can be obtained and maintained in one head, not necessarily the same over time.

**Increase of Productivity by Functional Languages.** It is quite obvious that we should use formal methods eventually to produce our own tools; but is this realistic for really large systems? and during the initial bootstrapping phase of developing the first tools that allow scaling up? The answer is to use programming languages whose

- high-level aids self-documentation and maintenance,
- features for genericity instigate compactness and reuse,
- modularisation, information-hiding and inheritance features encourage structuring,
- static properties such as strong typing allow comprehensive checks,
- implementation permits separate compilation and, ideally, interpretation.

Our experience has been best with functional programming languages so far; we estimate the increase of productivity over, say, C, to a factor of 3. Without them, the development of large, non-trivial tools over a period of several years would have been impossible in an academic environment. TAS and IsaWin are extensions of Isabelle and comprise about 25k lines of SML; the graph visualisation system *daVinci* with thousands of users was developed by Fröhlich and Werner (1994, see also (Fröhlich 1997), and cf. fig. 7) over a period of 5 years comprising about 35k lines of a func-

tional language developed at Bremen, plus about 10k lines of C for interfacing; the tool integration framework of the UniForM Workbench, developed (see below), contains about 50k lines of Haskell.

**Integration of Tools in the UniForM Workbench** is described in detail in a companion paper (Karlsen 1998a; see also 1998b). Control and data integration is provided by the Subsystem Interaction Manager; based on the UniForM Concurrency Toolkit, tools interact like communicating concurrent agents and are, in general, loosely coupled by intermittent adaptors (cf. Karlsen 1997a, 1997b). The Repository Manager is a graphical interface (using *daVinci*) to a public domain version of the industry standard Portable Common Tool Environment and provides version and configuration control, etc. (cf. (PCTE 1994), (H-PCTE 1996), and (Karlsen and Westmeier 1997)).

The User Interaction Manager provides presentation integration, incorporating interfaces to *daVinci* and its extension Forest, a WWW-browser, and Tcl/Tk for window management. In particular the latter two become much more manageable and homogeneous by encapsulation into a typed, high-level interface in Haskell.

Haskell is the internal integration language; thus even higher-order objects and processes can be transmitted as objects. External tools are wrapped into a Haskell interface; however, we plan an adaptation of the Interface Definition Language of the industry standard CORBA to Haskell that will then shortly open more possibilities to integrate tools in, say, C, C++, or Java.

**The Most Promising Architectures for Development Tools** are those that avoid self-containment and allow integration with others. The possibility for control and data integration of a tool as an "abstract data type" is the most important (and not obvious since the tool may e.g. not allow remote control and insist on call-backs); integration of persistent data storage in a common repository is next (this may require export and import w.r.t. local storage); presentation integration with the same user interface is last - in fact it is most likely that the tool has its own graphical user interface. However, interactive Posix tools usually have a line-oriented interface that can easily be adapted (Karlsen 1997b).

This way, a graphical interface to HUGS was developed in a matter of weeks. Isabelle, IsaWin and TAS have been integrated, and a Z-Workbench with various tools has been instantiated from the UniForM Workbench (Lüth et al. 1998a, 1998b).

## 6 Conclusion

Formal methods are on the verge of becoming competitive in an industrial context, they are even cost-effective now when we consider their use in automated testing or deadlock detection for software that has been developed in a conventional way.
We have presented the following theses:

- Combination of various methods is essential; more basic research is needed here.
- Treatment of real-time and hybrid systems, in particular, is just emerging.
- Testing benefits greatly from the use of Formal Methods.
- Model checking and interactive verification of properties both are necessary.

- Synthesis by transformation has great potential over invent-and-verify.
- Reuse of formal developments is much more powerful than reuse of products.
- Tools for process models should support tailoring and generation of documents.
- The success of formal methods depends on the usability of support tools.
- Development environments should integrate tools in a loosely coupled way.

## 7 References


Amthor, P., Dick, S. (1997): Test eines Bordcomputers für ein dezentrales Zugsteuerungssystem unter Verwendung des Werkzeuges VVT-RT. 7. *Kolloquium Software-Entwicklung Methoden, Werkzeuge, Erfahrungen: Mächtigkeit der Software und ihre Beherrschung*, Technische Akademie Esslingen.

Astesiano, E. and Cerioli, M. (1994): Multiparadigm Specification Languages: a First Attempt at Foundations, In: C.M.D.J. Andrews and J.F. Groote (eds.), Semantics of Specification Languages (SoSl'93), Workshops in Computing, pp. 168-185. Springer.

Blank Purper, C., Westmeier, S. (1998): A Graphical Development Process Assistant for Formal Methods. In: *Proc. VISUAL'98 (short papers)*, at ETAPS'98, Lisbon. `http://www.tzi.de/~uniform/gdpa`

Buth, B., Kouvaras, M., Peleska, J., Shi, H. (1997): Deadlock Analysis for a Fault-Tolerant System. In Johnson, M. (ed.): Algebraic Methodology and Software Technology. *AMAST'97*. LNCS 1349. Springer, pp. 60-75.

Buth, B., Peleska, J., Shi, H. (1998): Combining Methods for the Livelock Analysis of a Fault-Tolerant System. (submitted to AMAST'98)

Cerioli, M., Haxthausen, A., Krieg-Brückner, Mossakowski, T. (1997): Permissive Order-sorted Partial Logic in CASL. *AMAST'97*. LNCS 1349. Springer.

CoFI: The Common Framework Initiative for Algebraic Specification. `http://www.brics.dk/Projects/CoFI`

Diaconescu, R. (1998): Extra Theory Morphisms for Institutions: logical semantics for multi-paradigm languages. *J. Applied Categorical Structures* (to appear).

Dierks, H. (1997): PLC-Automata: A New Class of Implementable Real-Time Automata. Proc. ARTS'97. *LNCS 1231*, pp. 111-125. Springer.

Fischer, C. (1997): CSP-OZ: A Combination of Object-Z and CSP. In: *Proc. Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*.

Fröhlich, M. (1997): Inkrementelles Graphlayout im Visualisierungssystem *daVinci*. Dissertation. *Monographs of the Bremen Institute of Safe Systems 6*, Aachen, Shaker.

Fröhlich, M., Werner, M. (1994): The interactive Graph-Visualization System *daVinci* – A User Interface for Applications. Informatik Bericht Nr. 5/94, Universität Bremen. updated documentation: `http://www.tzi.de/~daVinci`

H-PCTE (1996): The H-PCTE Crew: H-PCTE vs. PCTE, Version 2.8, Universität Siegen.

Haxthausen, A. E., Peleska, J. (1998): Formal Development and Verification of a Distributed Railway Control System. In *Proc. 1st FMERail Workshop*, Utrecht, The Netherlands, (to appear).

Hoffmann, B., Krieg-Brückner, B. (eds.) (1993): *PROgram Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System*. LNCS 680. Springer. `http://www.tzi.de/~prospectra`

Karlsen, E. W. (1997a): The UniForM Concurrency ToolKit and its Extensions to Concurrent Haskell. In: O'Donnald, J. (ed.): *GFPW'97, Glasgow Workshop on Functional Programming '97*, Ullapool.

Karlsen, E. W. (1997b): Integrating Interactive Tools using Concurrent Haskell and Synchronous Events, In *ClaPF'97, 2nd Latin-American Conference on Functional Programming*, La Plata, Argentina.

Karlsen, E. W. (1998a): The UniForM WorkBench - a Higher Order Tool Integration Framework. In: *International Workshop on Current Trends in Applied Formal Methods*. LNCS. Springer (*this volume*).

Karlsen, E. W. (1998b): Tool Integration in a Functional Setting. Dissertation. Universität Bremen. 364pp. *(to appear)*.

Karlsen, E. W., Westmeier, S. (1997): Using Concurrent Haskell to Develop User Interfaces over an Active Repository. In *IFL'97, Implementation of Functional Languages 97*, St. Andrew, Scotland, September 10-12, 1997. LNCS 1467. Springer.

Kolyang (1997): HOL-Z, An Integrated Formal Support Environment for Z in Isabelle/HOL. Dissertation. *Monographs of the Bremen Institute of Safe Systems 5*, Aachen, Shaker.

Kolyang, Lüth, C., Meyer, T., Wolff, B. (1997): TAS and IsaWin: Generic Interfaces for Transformational Program Development and Theorem Proving. In Bidoit, M. and Dauchet, M. (eds.): *Theory and Practice of Software Development '97*. LNCS 1214. pp. 855-859. Springer.

Kolyang, Santen, T., Wolff, B. (1996a): A Structure Preserving Encoding of Z in Isabelle/HOL. In *Proc. Int'l Conf. on Theorem Proving in Higher Order Logic* (Turku). LNCS 1125. Springer. `http://www.tzi.de/~kol/HOL-Z`

Kolyang, Santen, T., Wolff, B. (1996b): Correct and User-Friendly Implementations of Transformation Systems. In: Gaudel, M.-C., Woodcock, J. (eds.): *FME'96: Industrial Benefit and Advances in Formal Methods*. LNCS 1051, pp. 629-648. Springer.

Krieg-Brückner, B. (1996): Seven Years of COMPASS. In: Haveraaen, M., Owe, O., Dahl, O.-J. (eds.): *Recent Trends in Data Type Specification.* Proc. 11th ADT/COMPASS Workshop (Oslo 1995). LNCS 1130, pp. 1-13. Springer. `http://www.tzi.de/~compass`

Krieg-Brückner, B., Peleska, J., Olderog, E.-R., Balzer, D., Baer, A. (1996): UniForM, Universal Formal Methods Workbench. in: Grote, U., Wolf, G. (eds.): *Statusseminar des BMBF: Softwaretechnologie.* Deutsche Forschungsanstalt für Luft- und Raumfahrt, Berlin 337-356. See also `http://www.tzi.de/~uniform`

Lankenau, A., Meyer, O., Krieg-Brückner, B. (1998): Safety in Robotics: The Bremen Autonomous Wheelchair. In: *Proc. AMC'98, 5th Int. Workshop on Advanced Motion Control*, Coimbra, Portugal 1998. ISBN 0-7803-4484-7, pp. 524-529.

Lüth, C. (1997): Transformational Program Development in the UniForM Workbench. *Selected Papers from the 8th Nordic Workshop on Programming Theory*, Oslo, Dec. 1996. Oslo University Technical Report 248.

Lüth, C. and Wolff, B. (1998): Functional Design and Implementation of Graphical User Interfaces for Theorem Provers. (to appear in *Journal of Functional Programming*).

Lüth, C., Karlsen, E. W., Kolyang, Westmeier, S., Wolff, B. (1998a): Tool Integration in the UniForM WorkBench. In Berghammer, B., Buth, B., Berghammer, R., Peleska, J. (eds.): Tools for System Development and Verification. Workshop, Bremen, July 1996. *Monographs of the Bremen Institute of Safe Systems 1,* Aachen, Shaker.

Lüth, C., Karlsen, E. W., Kolyang, Westmeier, S., Wolff, B. (1998b): HOL-Z in the UniForM WorkBench - a Case Study in Tool Integration for Z. *Proc. ZUM'98, 11th International Conference of Z Users*, LNCS 1493, pp. 116-134. Springer.

Lüth, C., Shi, H., Krieg-Brückner, B. (1999): Abstraction in Transformational Developments. (submitted for publication).

Lüth, C., Westmeier, S., Wolff, B. (1996): sml_tk: Functional Programming for Graphical User Interfaces. Informatik Bericht Nr. 8/96, Universität Bremen. `http://www.tzi.de/~cxl/sml_tk`

Mossakowski, T. (1996): Representations, Hierarchies and Graphs of Institutions. Dissertation. *Monographs of the Bremen Institute of Safe Systems 2*, Aachen, Shaker.

Mossakowski, T. (1998): Translating other specification languages to CASL. *Recent Trends in Algebraic Development Techniques*. WADT'98, Lisbon. LNCS. (to appear)

Mossakowski, T., Kolyang, Krieg-Brückner, B. (1998a): Static Semantic Analysis and Theorem Proving for CASL. In Parisi-Pressice, F. (ed.): *Recent Trends in Algebraic Development Techniques*. WADT'97, LNCS 1376, pp. 333-348. Springer.

Mossakowski, T., Tarlecki, A., Pawlowski, W. (1997): Combining and Representing Logical Systems, In . Moggi, E. and Rosolini, G. (eds.): *Category Theory and Computer Science*, 7th Int. Conf. LNCS 1290, pp. 177-196. Springer.

Mossakowski, T., Tarlecki, A., Pawlowski, W. (1998b): Combining and Representing Logical Systems Using Model-Theoretic Parchments. In Parisi-Pressice, F. (ed.): *Recent Trends in Algebraic Development Techniques*. WADT'97, LNCS 1376, pp. 349-364. Springer.

Mosses, P. (1997): CoFI: The Common Framework Initiative for Algebraic Specification and Development. In Bidoit, M. and Dauchet, M. (eds.): *Theory and Practice of Software Development '97*. LNCS 1214, pp 115-137. Springer.

Paulson, L. (1995): *Isabelle: A generic theorem prover*. LNCS 828. Springer.

PCTE (1994): European Computer Manufacturers Association: *Portable Common Tool Environment (PCTE), Abstract Specification*, 3rd edition, ECMA-149. Geneva.

Peleska, J. (1996): Formal Methods and the Development of Dependable Systems. Bericht 1/96, Universität Bremen, Fachbereich Mathematik und Informatik, 72p. `http://www.tzi.de/~jp/papers/depend.ps.gz`

Peleska, J., Siegel, M. (1996): From Testing Theory to Test Driver Implementation. in: M.-C. Gaudel, J. Woodcock (eds.): *FME'96: Industrial Benefit and Advances in Formal Methods*. LNCS 1051. Springer, pp. 538-556.

Peleska, J., Siegel, M. (1997): Test Automation of Safety-Critical Reactive Systems. *South African Computer Jounal 19*, pp. 53-77.

Sernadas, A., Sernadas, C., Caleiro, C. (1998a): Fibring of logics as a categorial construction. *Journal of Logic and Computation8 (10)*, pp. 1-31.

Sernadas, A., Sernadas, C., Caleiro, C., Mossakowski, T. (1998b): Categorical Fibring of Logics with Terms and Binding Operators. In Frontiers of Combining Systems - FroCoS'98. Kluwer Academic Publishers. To appear in Applied Logic Series.

Tapken, J. (1997): Interactive and Compilative Simulation of PLC-Automata. In: Hahn, W. and Lehmann, A. (eds.): *Simulation in Industry*, ESS'97. SCS, pp. 552-556.

Tapken, J. (1998): MOBY/PLC – A Design Tool for Hierarchical Real-Time Automata. System Demo. In: Astesiano, E. (ed.): *Proc. First Int'l Conf. on Fundamental Approaches to Software Engineering, FASE'98*, at ETAPS'98, Lisbon. LNCS. pp326-330. Springer.

Tarlecki, A. (1996): Moving between logical systems. In M. Haveraaen and O. Owe and O.-J. Dahl (eds.): *Recent Trends in Data Type Specifications.* 11th Workshop on Specification of Abstract Data Types. *LNCS 1130*, pp. 478-502. Springer.

Tej, Haykal (1999): HOL-CSP: Mechanised Formal Development of Concurrent Processes. Dissertation. *Monographs of the Bremen Institute of Safe Systems*. (forthcoming)

Tej, H. and Wolff, B. (1997): A Corrected Failure-Divergence Model for CSP in Isabelle / HOL. *Formal Methods Europe, Proc. FME'97*, Graz. LNCS 1313, pp. 318-337. Springer.

Urban, G., Kolinowitz, H.-J., Peleska, J. (1998): A Survivable Avionics System for Space Applications. in *Proc. FTCS-28, 28th Annual Symposium on Fault-Tolerant Computing*, Munich, Germany, 1998.

V-Model (1997): *Development Standard for IT Systems of the Federal Republic of Germany.* General Directives 250: Process Lifecycle; 251: Methods Allocation.

Zhou, C., Hoare, C.A.R., Ravn, A.P. (1992): A Calculus of Durations. *Information Processing Letters 40(5)* pp. 269-276.