The UniForM Workbench, a Universal Development Environment for Formal Methods

Bernd Krieg-Brückner[1], Jan Peleska[1], Ernst-Rüdiger Olderog[2], Alexander Baer[3]

[1] Bremen Institute of Safe Systems, University of Bremen, PBox 330440, D-28334 Bremen
bkb@Informatik.Uni-Bremen.DE, jp@Informatik.Uni-Bremen.DE
[2] University of Oldenburg, PBox 2593, D-26111 Oldenburg
Olderog@Informatik.Uni-Oldenburg.DE
[3] INSY, Marzahnerstr. 34, D-13053 Berlin
insy_abaer@compuserve.com

Abstract. The UniForM Workbench supports combination of Formal Methods (on a solid logical foundation), provides tools for the development of hybrid, real-time or reactive systems, transformation, verification, validation and testing. Moreover, it comprises a universal framework for the integration of methods and tools in a common development environment. Several industrial case studies are described.

1 Introduction

The UniForM Workbench (Universal Formal Methods Workbench, cf. [K+96, K+99, Kri99]) has been developed by the Universities of Bremen and Oldenburg, and Elpro, Berlin, funded by the German Ministry for Education and Research, BMBF.

Formal Methods are used in modelling, using a mathematically well-founded specification language, proving properties about a specification and supporting correct development. The need 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.

It is unrealistic to expect 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

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1186-1205, 1999.
© Springer-Verlag Berlin Heidelberg 1999

(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) enable verification of particular properties, e.g. adherence to a security model, absence of deadlocks or satisfaction of performance requirements.

Another issue is the correct combination and integration of tools to support Formal Methods. Tools invariably differ in the exact language or semantics they support; the tool combination has to realize a correct combination of the resp. methods.

2 Combination of Methods

2.1 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 [VMOD] 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.

Up to now, tool support for working with the V-Model has mostly been provided by stand-alone project management components, facilitating the document production process for the project manager. In the UniForM project, we have adopted a different approach to V-Model utilisation: Formally speaking, the V-Model is a generic specification for the system development process. Tailoring the V-Model for a particular enterprise means instantiating this development process specification by determining

- the products (specifications, code, hardware, tests, proofs etc.) to be created,
- the activities and people responsible for each product,
- the methods to be used for each development step, and
- the tools to be used for application of these methods.

We are convinced that this instantiation process is best performed in the development environment itself, so that the tailoring process will not only have project management documents as output but simultaneously configure the Workbench for the specific configuration to be used in the development project.

This approach is presently implemented by Purper [BW98, Pur99a, b] in 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 tool support items 1 to 4 above; cf. also fig.1. Integration into a development environment such as the UniForM Workbench allows the coordination with its methods and tools (item 3). Tools themselves can generate development documents in conformance with the V-model (cf. item 4), such as the development history of fig. 6.

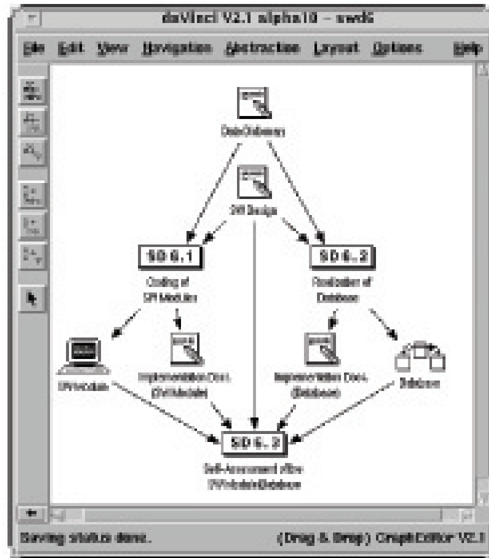

Fig. 1. *Example of a V-Model Process Graph as supported by the* UniForM Workbench

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. PLC-Automata in section 2.2 and 3.1).

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 [LMK98].

2.2 Combination of Formal Methods

Combinations of Formal Methods are by no means easy to achieve. The need for research 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. [AC94, Tar96, Dia98]); approaches for reasoning about correct composition of the logics capturing the semantics "in the small" (see e.g. [Mos96, Mos99b, MTP97, MTP98, SSC98, S+98]) introduce notions such as *embedding, translating* one formalism to another, *combination* of two formalisms, or *projecting* to either from the combination.

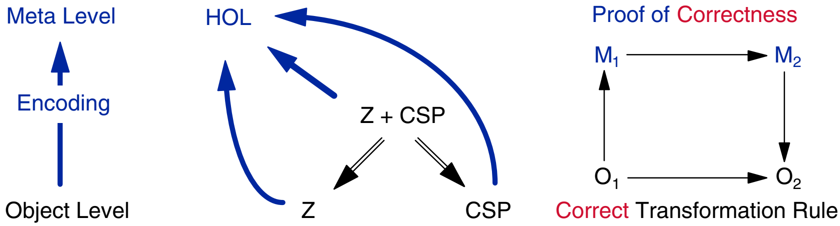

Fig. 2. Semantic Representation in UniForM

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 realized in the logical framework Isabelle [Pau95]. Fig. 2 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 3.3), and, as a basis for transformational development (cf. section 3.4), 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. [KSW96a, KSW96b, K+97, Kol97, L+98]) 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 [TW97, Tej99]. 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 [MKK98].

Reactive Real-Time Systems. The first instantiation of UniForM has been for Z and CSP since these are considered to be rather mature and have been successfully applied to industrial cases. At the moment, we are working on methods ("structural transformations") to project not only from Z+CSP (actually Object-Z, cf. [Fis97, FS97]), 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. 3. 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 3.4).

Combination of CSP and Object-Z. Whereas CSP is well suited for the description of communicating processes, Object-Z is an object based specification method for data, states and state transformations. Motivated by previous work at Oldenburg in

the ESPRIT Basic Research Action ProCoS (Provably Correct Systems) [ProCoS], a combination of both methods into the specification language CSP-OZ has been proposed in [Fis97, FS97]. In CSP-OZ the process aspects are described using CSP and the data aspects using Object-Z. A specific achievement is the simple semantics of the combination which is based on two ideas:

- the embedding of Object-Z into the standard semantic model of CSP, the so-called failures/divergences model [Ros97]
- the semantic definition of the combination by the synchronous, parallel composition of the CSP part and the Object-Z part of a CSP-OZ specification.

Thus to each CSP-OZ specification a semantics in the failures/divergences model is assigned. As a consequence the concept of refinement of this model is also applicable to CSP-OZ. It has been shown that both process refinement of the CSP part and data refinement of the Object-Z part yield refinement of the whole CSP-OZ specification [Hal97]. Thus FDR (failures/divergences refinement), a commercially available model checker for CSP [FDR96], can also be applied to CSP-OZ specifications.

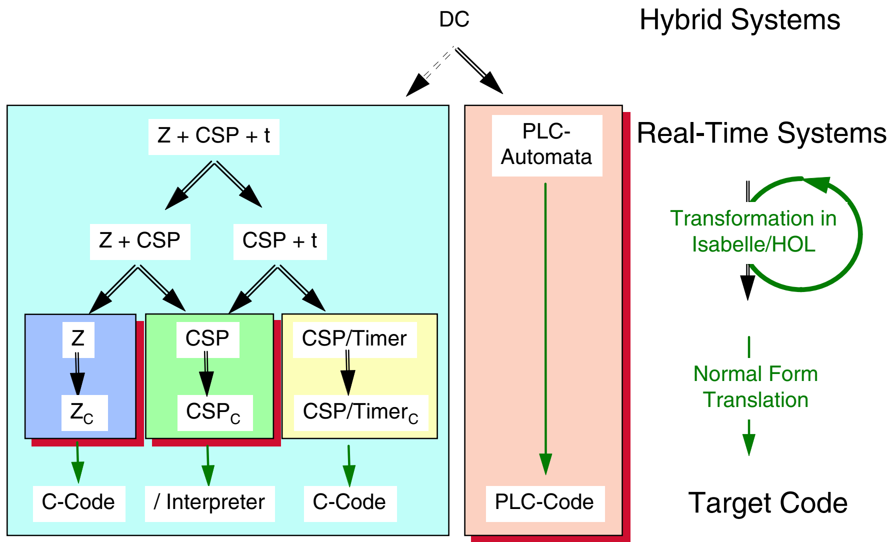

Fig. 3. *Method Combination in* UniForM

Combination of PLCs and Duration Calculus. For the specification of time critical aspects of computer systems, the Duration Calculus (DC for short, cf. [ZHR92]) was chosen from the start of the UniForM project. DC is intended for a formalization of high-level requirements.

On the lowest level, Programmable Logic Controllers (PLCs for short) were considered because they are simple devices that are widespread in control and automation technology. A PLC interacts with sensors and actuators in a cyclic manner. Each cycle consists of three phases: an input phase where sensor values are read and stored in local variables, a state transformation phase where all local variables are updated according to the stored program, and an output phase where the values of some of the

local variables are output to the actuators. Real-time constraints can be implemented on PLCs with the help of timers that can be set and reset during the state transformation phase. The reaction time of a PLC depends on the cycle time.

One of the challenges of the UniForM project was to bridge the gap between Duration Calculus and PLCs in such a way that the correctness of the PLC software can be proven against the requirements formalised in DC. One of the discoveries in the UniForM project was that the behaviour of PLCs can very well be modelled using a novel type of automaton called *PLC-Automaton* [Die97], cf. fig. 4. The semantics of PLC-Automata describes the cyclic behaviour of a PLC; it is defined in terms of the Duration Calculus. Thus PLC-Automata represent a combination of the concept of PLC with DC. This enables us to integrate PLC-Automata into a general methodology for the design of real-time systems based on DC [Old98].

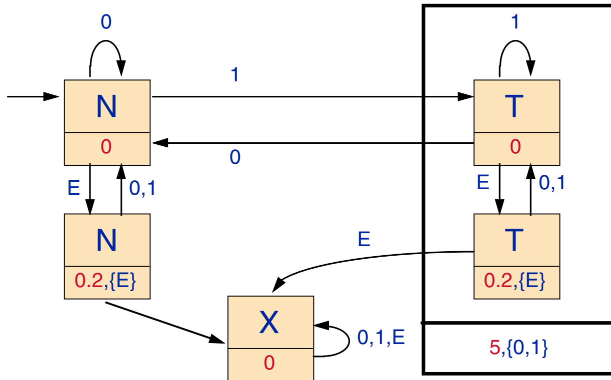

Fig. 4: *PLC-Automaton*

The CoFI 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. [Kri96]), 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. 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. It has a complete formal semantics. CASL is a rather powerful and general specification language for first-order logic specifications with partial and total functions, predicates, subsorting, and generalized overloading [CoFI, C+97, Mos97]. 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 [Mos99a]; this aspect is crucial for its intended impact. Various parsers exist; the first prototype implementation in the UniForM

Workbench [MKK98] comprises static semantic analysis for basic specifications and theorem proving in Isabelle; it will be the basis for transformational development.

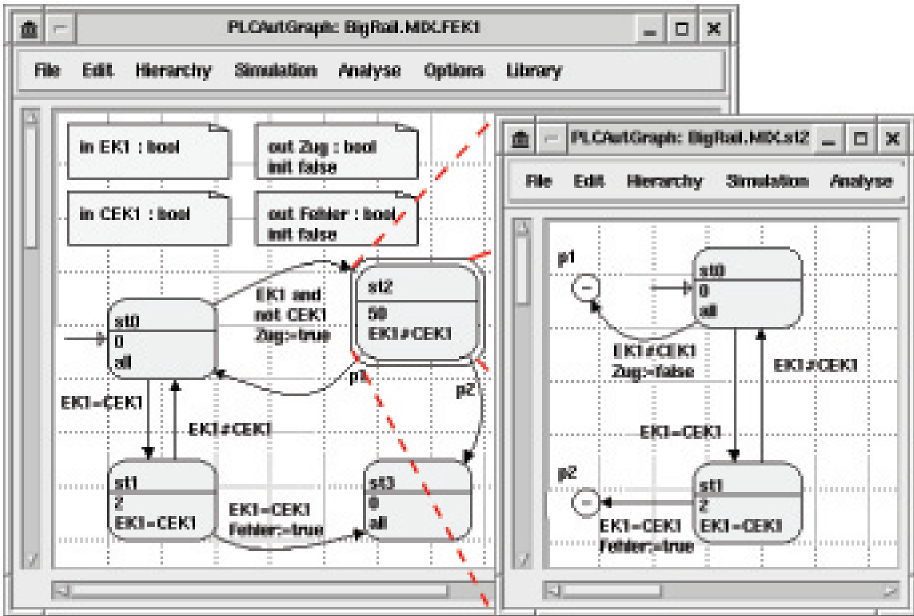

Fig. 5: *The Moby/PLC tool for the development of PLC-Automata*

3 Tools for Development

3.1 Development of PLC Software

At the University of Oldenburg a tool called Moby/PLC was designed and implemented that supports the work with PLC-Automata [DT98], see fig. 5. The tool comprises the following components:

- a graphical editor for drawing PLC-Automata
- a simulator for networks of PLC-Automata
- a compiler for generating PLC code in ST (Structured Text), a dedicated programming language for PLCs
- an algorithm for the static analysis of real-time constraints
- compilers for generating input for the real-time model checkers UPPAAL [B+95] and KRONOS [D+96]
- a synthesis algorithm for generating PLC-Automata from specifications written in a subset of Duration Calculus, so-called DC Implementables.

3.2 Tools for CSP-OZ

For the combined specification language CSP-OZ a graphical editor called Moby/OZ was developed. It is based on the same class library as the Moby/PLC tool. The editor enables the user to perform type checking using the Object-Z type checker "wizard".

3.3 Verification

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 domains, application developers become increasingly aware of the importance of methods *guaranteeing* correctness w.r.t. a formal specification requirements, be it by the invent-and-verify paradigm, synthesis or transformation.

Abstraction to Verify Special Properties. In [B+97, BPS98, UKP98], a technique for abstracting from an existing program to verify the absence of deadlocks and livelocks was developed. It was applied successfully to more than 25k 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 [FDR96].

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

Model-Checking is a very important technique in practice. The FDR tool [FDR96] 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 section 5.3). 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, the FDR tool can be used within the interactive proof tool.

Moreover, the experience of [HP98] when solving the train control problem in general (cf. also section 5.3) 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.

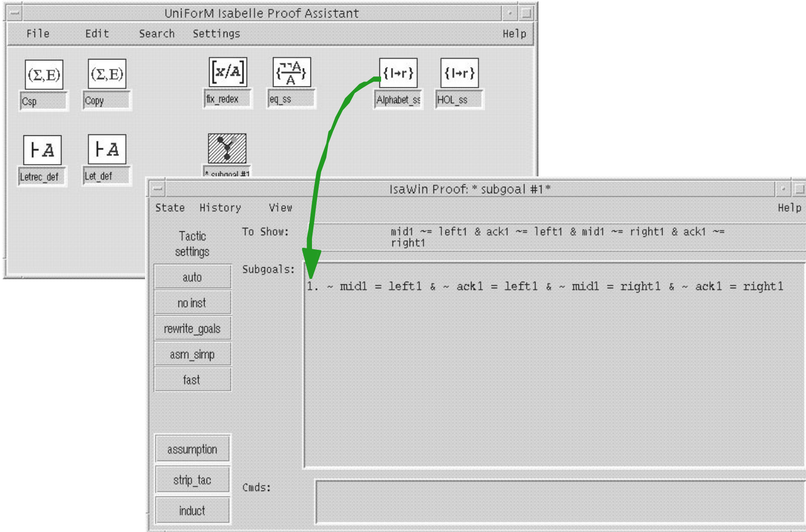

Fig. 6. *The* Isabelle *Proof Assistant* IsaWin *in* UniForM

A Window to Isabelle. The UniForM Workbench makes extensive use of the generic theorem prover Isabelle [Pau95], 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 [K+97, LW98]. Objects such as theories, substitutions, proof rules, simplification sets, theorems and proofs are typed (cf. fig. 6); 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.

3.4 Development by Transformation

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. 7, see [LW98, L+98, L+99]). Like Isabelle, it is implemented in Standard ML; sml_tk [LWW96] is a typed interface in

SML to Tcl/Tk; on top, the generic user interface GUI provides the appearance of fig. 6 and fig. 8. 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.

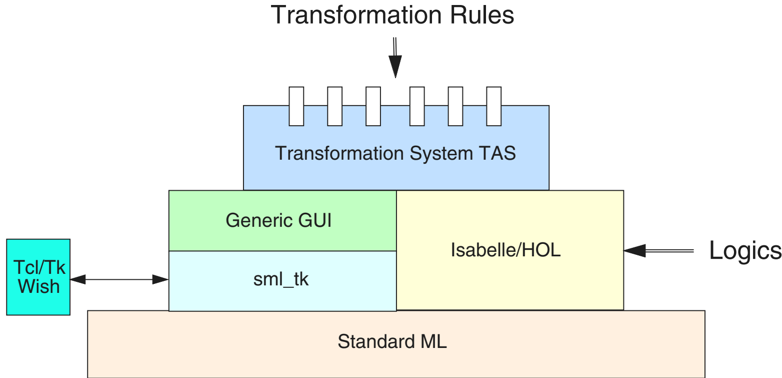

Fig. 7. *Architecture of* TAS*, the* UniForM *Transformation System*

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. 2.

TAS, the UniForM Transformation System. TAS may be parametrized by a logic (e.g. semantic representation of Z, CSP or CASL) at the Isabelle level, and by transformation rules at the level of TAS itself, cf. fig. 7 [Lüt97, L+99]. 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. 8). 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 example shows the development of a communication protocol with send / receive buffers by a sequence of transformations in CSP.

The functionality of TAS subsumes that of a forerunner, the PROSPECTRA system [HK93]. However, the basis of Isabelle allows a more compact, more flexible and more powerful realisation: parametrization 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.

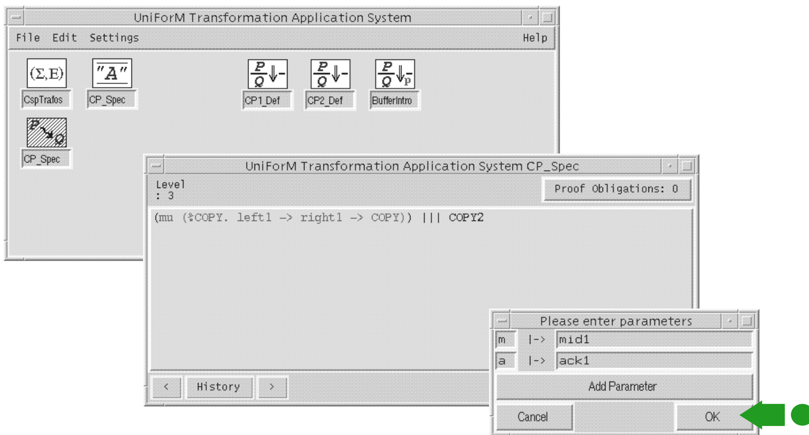

Fig. 8. Application of a Parametrized Transformation Rule

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: initial and current specification, proof obligations, and development history.

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.

3.5 Validation, Verification, and Test Environment for Reactive Real-Time Systems

For real-world large-scale systems, complete formal development is still unrealistic: The amount of code implementing the application, operating system, drivers and firmware is simply too large to admit complete formal treatment. Furthermore, many correctness aspects of reactive systems depend on the interaction of software and

hardware, and the number of different hardware components is too high to allow for the creation of formal behavioural models of these components. As a consequence, our recommendation to the Formal Methods practitioner is as follows:

- Try to develop the logical concepts (communication flow, control algorithms, data transformation algorithms etc.) in a formal way, in order to avoid logical flaws creeping into system design and implementation.
- Perform formal code development as far as possible, with emphasis on the critical modules of the system, otherwise use testing and inspection techniques.
- Use automated testing to check the proper integration of software and hardware. To support such an approach, the VVT-RT (Verification, Validation and Test for Real-Time Systems) tool kit is currently integrated into the UniForM Workbench:

Verification, Validation, and Testing. The methodology and tool kit VVT-RT [Pel96, PS96, PS97] allows *automatic* testing and verification and validation of (test) specifications. 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 is the automatic test evaluation component of the tool kit: In practice, the execution of real-time tests will lead to thousands of lines of timed traces recording the occurrence of interleaved inputs and outputs over time. Manual inspection of these traces would be quite impossible. Instead, VVT-RT performs automatic evaluation of timed traces against a binary graph representation of the formal specification. This approach is very cost-effective. It has been applied successfully to one of the case studies of UniForM, a control computer on board of a train for railway control (see section 5.3), and to an electric power control component of a satellite developed by OHB, Bremen [Mey98, SMH99].

4 Universal Development Environment

The UniForM Workbench is an open ended tool integration framework for developing (formal) software development environments from the basis of pre-fabricated off-the-shelf development tools. The Workbench uses Concurrent Haskell as its central integration language, extended with a higher order approach to event handling akin to the one found in process algebras. Integration can therefore be done at a high level of abstraction, which combines the merits of functional programming with state-of-the-art concurrent programming languages.

The Workbench provides support for data, control and presentation integration as well as utilities for wrapping Haskell interfaces around existing development tools. It views the integrated Workbench as a reactive (event driven) system, with events amounting to database change notifications, operating system events, user interactions and individual tool events. The unique feature of the Workbench is that it provides a uniform and higher order approach to event handling, which improves on traditional approaches such as callbacks, by treating events as composable, first class values.

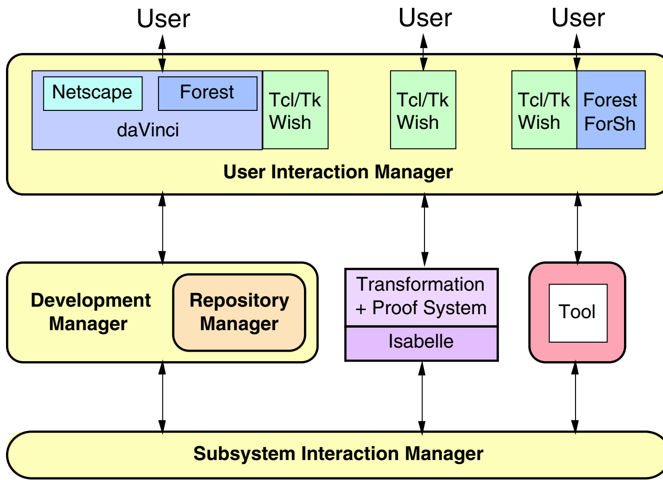

Fig. 9: *System Architecture of the* UniForM Workbench

Integration of Tools in the UniForM Workbench is described in detail in [Kar99] (see also [Kar98]), cf. fig. 9. *Control integration* is provided by the Subsystem Interaction Manager; based on the UniForM Concurrency Toolkit, tools interact in a fine grained network of communicating concurrent agents and are, in general, loosely coupled by intermittent adaptors (cf. [Kar97a, Kar97b]). The Repository Manager [KW97] takes care of *data integration* with an interface to a public domain version of the industry standard Portable Common Tool Environment [PCTE94, HPCTE] and provides version and configuration control, etc. with a graphical interface (using daVinci; cf. also fig. 10).

The User Interaction Manager provides *presentation integration*, incorporating interfaces to daVinci (see [FW94, Frö97], and cf. fig. 7 and fig. 10) 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; we are working on an adaptation of the Interface Definition Language of the industry standard CORBA to Haskell that will shortly open more possibilities to integrate tools in, say, C, C++, or Java.
Architectures for development tools should 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 [Kar97b].

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 (L+98), cf. fig. 10.

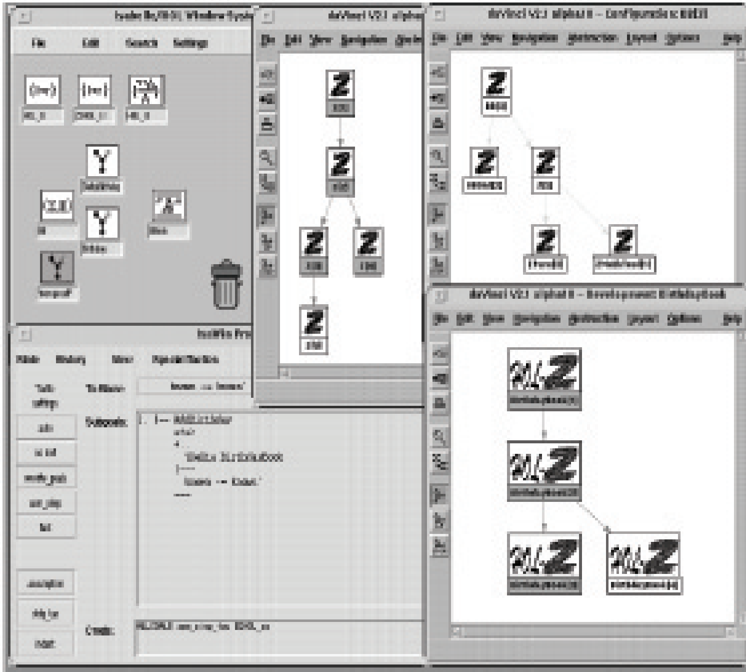

Fig. 10: Z-Workbench

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 at the moment for really large systems? 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 (in number of lines, backed by empirical evidence). 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 was developed by Fröhlich and Werner [FW94, Frö97] over a period of 5 years comprising about 35k lines of a functional language developed at Bremen, plus about 10k lines of C for interfacing; the tool integration framework of the UniForM Workbench was developed almost entirely by Karlsen [Kar99] in about 50k lines of Haskell.

5 Case Studies

5.1 Control of a Single Track Segment

In close cooperation with the industrial partner Elpro, a case study "control of a single track segment" was defined. The problem concerns the safety of tram traffic on a segment where only a single track is available, see fig. 11. Such a bottle-neck can occur for example during repair work. The task is to control the traffic lights in such a way that collisions of trams driving in opposite direction is avoided and that certain general traffic rules for trams are obeyed. Real-time requirements occur locally at the sensor components ES1, CS1, LS1, ES2, CS2, LS2 near the track.

The methodology of PLC-Automata was applied to this case study. Starting from informal requirements of the customer, in this case the Berlin traffic company, a network consisting of 14 PLC-Automata was constructed using the Moby/PLC tool as part of the UniForM Workbench [DT98, Die97]. With Moby/PLC the whole network could be simulated [Tap97]. Essential safety properties were proven, for example that at most one direction of the single track segment will have a green traffic light. Then the network of PLC-Automata was compiled into 700 lines of PLC code in the programming language ST (Structured Text), which can be distributed over the PLCs as indicated in fig. 11.

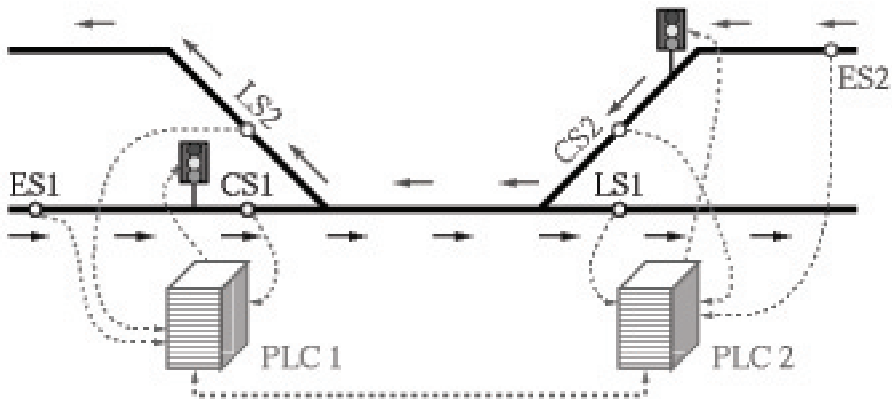

Fig. 11: *Tram Control*

5.2 Control of Jena Steinweg

After this first successful experiment with Moby/PLC, the Oldenburg group was challenged by Elpro to attack a more demanding case study where trams are allowed to drive into and out of the single track segment in many different ways. This control was actually implemented by Elpro in the city of Jena, hence the name. The complexity of this case study is due to the fact that the signalling of the traffic lights critically depends on the history of the last tram movements.

This case study could also be modelled with Moby/PLC as a network consisting of 110 PLC-Automata. While simulation still worked well, attempts to perform automatic verification of properties by translating the PLC-Automata into input for the real-time model checkers UPPAAL [B+95] and KRONOS [D+96] failed so far due to the complexity of the resulting timed automata. This complexity is caused by the fact that PLC automata take the cycle times of PLCs explicitly into account, in order to detect problems between communicating PLCs with different cycle times.

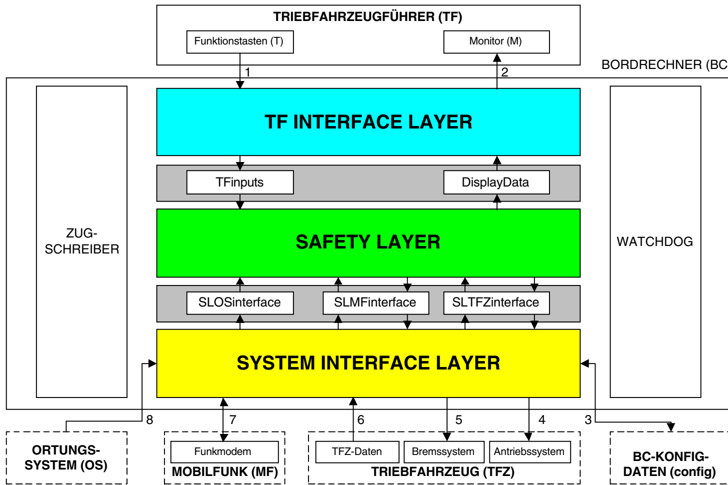

Fig. 12: Architecture of the On-Board Computer

5.3 On-Board Computer for Railway Control

Another case study was the development of a control computer on board of a train for railway control [AD97]. It is part of a distributed train and switching control system developed by Elpro, Berlin, where decentralised track-side safety-units control local points and communicate with trains via mobile phones. The whole system (with single tracks and deviations) has been modelled and specified with CSP in a student project at Bremen (cf. also [HP98] for a solution of the general train control problem).

The on-board computer has a layered architecture (see fig. 12). The TF INTERFACE LAYER communicates with the driver, the SYSTEM INTERFACE LAYER with a localization subsystem (e.g. GPS), a mobile phone subsystem and the train. The SAFETY LAYER contains all safety-relevant modules, determining the local state of the train, the requirements on the decentralized units on the track and their answers, finally leading to a decision about the permission to continue for the driver, or, alternatively, a forced-braking command to the train.

The design of the abstracts away from physical data formats in the concrete interfaces. The formal specification in CSP as an abstract transition system could be directly transliterated into an executable program that calls C^{++} functions of the interfaces [AD97, Pel96b].

The mobile phone based communication of the on-board computer with the track-side units in the SYSTEM INTERFACE LAYER is an example of the combination of CSP and OZ [Fis97, FS97], cf. sections 2.2 and 3.2.

The above case studies were done within the UniForM project; for other industrial applications of the UniForM Workbench cf. the verification of absence of deadlocks and livelocks for the International Space Station ([B+97, BPS98, UKP98], see section 3.3) or the automatic testing of an electric power control component of a satellite ([Mey98, SMH99], see section 3.5).

6 References

[AD 97] Amthor, P., Dick, S.: 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 (1997).

[AC94] Astesiano, E., Cerioli, M.: 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, Springer (1994) 168-185.

[BW98] Blank Purper, C., Westmeier, S.: A Graphical Development Process Assistant for Formal Methods. In: *Proc. VISUAL'98 (short papers),* at ETAPS'98, Lisbon (1998). `http://www.tzi.de/~uniform/gdpa`

[B+95] Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL -- a tool suite for automatic verification of real-time systems. Proc. 4th DIMACS Workshop: *Verification and Control of Hybrid Systems*. New Brunswick, New Jersey, 1995.

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

[BPS98] Buth, B., Peleska, J., Shi, H.: Combining Methods for the Livelock Analysis of a Fault-Tolerant System. In Haeberer, A.M. (ed.): *Algebraic Methodology and Software Technology, AMAST'98.* LNCS 1548. Springer (1999) 124-139 .

[C+97] Cerioli, M., Haxthausen, A., Krieg-Brückner, B., Mossakowski, T.: Permissive Subsorted Partial Logic in CASL. In Johnson, M. (ed.): *Algebraic Methodology and Software Technology, AMAST 97*, LNCS 1349, Springer (1997) 91-107.

[CoFI] *CoFI: The Common Framework Initiative for Algebraic Specification and Development.* `http://www.brics.dk/Projects/CoFI`

[Dia98] Diaconescu, R.: Extra Theory Morphisms for Institutions: logical semantics for multi-paradigm languages. *J. Applied Categorical Structures 6* (1998) 427-453.

[Die97] Dierks, H.: PLC-Automata: A New Class of Implementable Real-Time Automata. Proc. *ARTS'97*, LNCS 1231, Springer (1997) 111-125.

[DT98] Dierks, H., Tapken, J.: Tool-Supported Hierarchical Design of Distributed Real-Time Systems. *Euromicro Workshop on Real Time Systems,* IEEE (1998) 222-229.

[D+96] Dawsa, C., Olivero, A., Tripakis, S., Yovine, S.: The tool KRONOS. In: R. Alur, T.A. Henzinger, E.D. Sontag (Eds.): *Hybrids Systems III – Verfication and Control.* LNCS 1066*,* Springer, (1996).

[FDR96] Formal Systems Ltd.: *Failures Divergence Refinement.* User Manual and Tutorial Version 2.0. Formal Systems (Europe) Ltd. (1996).

[Fis97] Fischer, C.: CSP-OZ: A Combination of Object-Z and CSP. In H. Bowmann, J. Derrick (eds.): *Formal Methods for Open Object-Based Distributed Systems, FMOODS `97,* volume 2, Chapmann & Hall (1997) 423-438.

[FS97] Fischer, C., Smith, G.: Combining CSP and Object-Z: Finite or infinite trace-Semantics? *Proc. FORTE/PSTV 97*, Chapmann & Hall(1997) 503-518.

[Frö97] Fröhlich, M.: Inkrementelles Graphlayout im Visualisierungssystem daVinci. Dissertation. 1997. *Monographs of the Bremen Institute of Safe Systems 6,* ISBN 3-8265-4069-7, Shaker , 1998.

[FW94] Fröhlich, M., Werner, M.: The interactive Graph-Visualization System daVinci – A User Interface for Applications. Informatik Bericht Nr. 5/94, Universität Bremen, 1994. updated doc.: `http://www.tzi.de/~daVinci`

[Hal97] Hallerstede, S.: Die semantische Fundierung von CSP-Z. Diplomarbeit, Universität Oldenburg, 1997.

[HPCTE] The H-PCTE Crew: H-PCTE vs. PCTE, Version 2.8, Universität Siegen, 1996.

[HP98] Haxthausen, A. E., Peleska, J.: Formal Development and Verification of a Distributed Railway Control System. In *Proc. 1st FMERail Workshop*, Utrecht (1998).

[HK93] Hoffmann, B., Krieg-Brückner, B. (eds.): *PROgram Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System.* LNCS 680. Springer, 1993. `http://www.tzi.de/~prospectra`

[Kar97a] Karlsen, E.W.: The UniForM Concurrency ToolKit and its Extensions to Concurrent Haskell. In: O'Donnald, J. (ed.): *GFPW'97, Glasgow Workshop on Functional Programming '97*, Ullapool.

[Kar97b] Karlsen, E.W.: Integrating Interactive Tools using Concurrent Haskell and Synchronous Events. In *ClaPF'97, 2nd Latin-American Conference on Functional Programming,* La Plata, Argentina (1997).

[Kar98] Karlsen, E.W.: The UniForM Workbench - a Higher Order Tool Integration Framework. In: *Int'l Workshop on Current Trends in Applied Formal Methods*. LNCS. Springer (*to appear*).

[Kar99] Karlsen, E.W.: *Tool Integration in a Functional Setting*. Dissertation. Universität Bremen (1998) 364pp (to appear)

[KW97] Karlsen, E.W., Westmeier, S.: Using Concurrent Haskell to Develop User Interfaces over an Active Repository. In *IFL'97, Implementation of Functional Languages 97*, St. Andrew, Scotland. LNCS 1467. Springer (1997).

[Kol98] Kolyang: HOL-Z, An Integrated Formal Support Environment for Z in Isabelle/HOL. Dissertation, 1997. *Monographs of the Bremen Institute of Safe Systems 5,* ISBN 3-8265-4068-9, Shaker, 1998.

[KSW96a] Kolyang, Santen, T., Wolff, B.: A Structure Preserving Encoding of Z in Isabelle/HOL. In *Proc. Int'l Conf. on Theorem Proving in Higher Order Logic*. LNCS 1125. Springer (1996). `http://www.tzi.de/~kol/HOL-Z`

[KSW96b] Kolyang, Santen, T., Wolff, B.: 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 (1996) 629-648.

[Kri96] Krieg-Brückner, B.: Seven Years of COMPASS. In: Haveraaen, M., Owe, O., Dahl, O.-J. (eds.): *Recent Trends in Data Type Specification, LNCS 1130* (1996) 1-13.

[Kri99] Krieg-Brückner, B.: UniForM Perspectives for Formal Methods. In: *Int'l Workshop on Current Trends in Applied Formal Methods*. LNCS. Springer (*to appear*).

[K+96] 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. `http://www.tzi.de/~uniform`

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

[K+99] Krieg-Brückner, B., Peleska, J., Olderog, E.-R., Balzer, D., Baer, A: UniForM Workbench, Universelle Entwicklungsumgebung für Formale Methoden; Schlußbericht. 1998. *Monographs of the Bremen Institute of Safe Systems 9*. ISBN 3-8265-3656-8. Shaker, 1999.

[LMK98] Lankenau, A., Meyer, O., Krieg-Brückner, B.: 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.

[Lüt97] Lüth, C.: 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, May 1997.

[LW98] Lüth, C. and Wolff, B.: Functional Design and Implementation of Graphical User Interfaces for Theorem Provers. *J. of Functional Programming* (to appear).

[L+98] Lüth, C., Karlsen, E. W., Kolyang, Westmeier, S., Wolff, B.: HOL-Z in the UniForM Workbench - a Case Study in Tool Integration for Z. In J. Bowen, A. Fett,, M. Hinchey (eds.): *Proc. ZUM'98, 11th International Conference of Z Users,* LNCS 1493, Springer (1998) 116-134.

[L+99] Lüth, C., Tej, H., Kolyang, Krieg-Brückner, B.: TAS and IsaWin: Tools for Transformational Program Development and Theorem Proving. In J.-P. Finance (ed.): *Fundamental Approaches to Software Engineering (FASE'99, at ETAPS'99).* LNCS 1577. Springer (1999) 239-243. `http://www.tzi.de/~agbkb`

[LWW96] Lüth, C., Westmeier, S., Wolff, B.: sml_tk: Functional Programming for Graphical User Interfaces. Informatik Bericht Nr. 8/96, Universität Bremen. `http://www.tzi.de/~cxl/sml_tk`

[Mey98] Meyer, O.: Automated Test of a Power and Thermal Controller of a Satellite. In: Test Automation for Reactive Systems - Theory and Practice. Dagstuhl Seminar 98361, Schloss Dagstuhl, (1998).

[Mos96] Mossakowski, T.: Using limits of parchments to systematically construct institutions of partial algebras. In M. Haveraaen, O. Owe, O.-J. Dahl, eds.: *Recent Trends in Data Type Specification, LNCS 1130*, Springer (1996) 379-393.

[Mos97] Mosses, P.: CoFI: The Common Framework Initiative for Algebraic Specification and Development. In Bidoit, M., Dauchet, M. (eds.): *Theory and Practice of Software Development '97.* LNCS 1214, Springer (1997) 115-137.

[Mos99a] Mossakowski, T.: Translating OBJ3 to CASL: the Institution Level. In J. L. Fiadeiro (ed.): *Recent Trends in Algebraic Development Techniques*. 13th Int'l Workshop, WADT'98, Lisbon, Selected Papers. *LNCS 1589* (1999) 198-214.

[Mos99b] Mossakowski, T.: Representation, Hierarchies and Graphs of Institutions. Dissertation, Universität Bremen, 1996. Revised version. *Monographs of the Bremen Institute of Safe Systems 2,* ISBN 3-8265-3653-3, Shaker, 1999.

[MKK98] Mossakowski, T., Kolyang, Krieg-Brückner, B.: Static Semantic Analysis and Theorem Proving for CASL. In Parisi-Pressice, F. (ed.): *Recent Trends in Algebraic Development Techniques*. WADT'97, LNCS 1376, Springer (1998) 333-348.

[MTP97] Mossakowski, T., Tarlecki, A., Pawlowski, W.: Combining and Representing Logical Systems, In Moggi, E. and Rosolini, G. (eds.): *Category Theory and Computer Science,* 7th Int. Conf. LNCS 1290, Springer (1997) 177-196.

[MTP98] Mossakowski, T., Tarlecki, A., Pawlowski, W.: Combining and Representing Logical Systems Using Model-Theoretic Parchments. In Parisi-Pressice, F. (ed.): *Recent Trends in Algebraic Development Techniques*. WADT'97, LNCS 1376, Springer (1998) 349-364.

[Old98] Olderog, E.-R.: Formal Methods in Real-Time Systems. In *Proc. 10th EuroMicro Workshop on Real Time Systems*. IEEE Computer Sociery (1998) 254-263.

[Pau95] Paulson, L. C.: *Isabelle: A Generic Theorem Prover*. LNCS 828, 1995.

[PCTE94] European Computer Manufacturers Association: *Portable Common Tool Environment (PCTE), Abstract Specification*, 3rd ed., ECMA-149. Geneva, 1994.

[Pel96a] Peleska, J.: Formal Methods and the Development of Dependable Systems. Bericht 1/96, Universität Bremen, Fachbereich Mathematik und Informatik (1996) 72p. `http://www.tzi.de/~jp/papers/depend.ps.gz`

[Pel96b] Peleska, J.: Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. In: M.-C. Gaudel, J. Woodcock (eds.): *FME'96: Industrial Benefit and Advances in Formal Methods*. LNCS 1051 (1996) 39-59.

[ProCoS] He, J., Hoare, C.A.R., Fränzle, M., Müller-Olm, M., Olderog, E.-R., Schenke, M., Hansen, M.R., Ravn, A.P., Rischel, H.: Provably Correct Systems. In H. Langmaack, W.-P., de Roever, J., Vytopil (Eds.): *Formal Techniques in Real-Time and Fault-Tolerant Systems.* LNCS 863, Springer (1994).288–335.

[PS96] Peleska, J., Siegel, M.: 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 (1996) 538-556.

[PS97] Peleska, J., Siegel, M.: Test Automation of Safety-Critical Reactive Systems. *South African Computer Jounal 19* (1997) 53-77. `http://www.tzi.de/~jp/papers/sacj97.ps.gz`

[Pur99a] Purper, C.: GDPA: A Process Web-Center. *Proc. 2nd Workshop on Software Engineering over the Internet,* with ICSE'99, Los Angeles, 1999. `http://sern.cpsc.ucalgary.ca/ ~maurer/ICSE99WS/Program.htm`

[Pur99b] Purper, C.: An Environment to support flexibility in process standards. *Proc. 1st IEEE Conf. on Standardization and Innovation in Information Technology*. Aachen, 1999 (to appear).

[Ros97] Roscoe, A.W.: *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[SMH99] Schlingloff, H., Meyer, O., Hülsing, Th.: Correctness Analysis of an Embedded Controller. In *Data Systems in Aerospace, DASIA '99*, Lissabon (May 1999).

[SSC98] Sernadas, A., Sernadas, C., Caleiro, C.: Fibring of logics as a categorial construction. *Journal of Logic and Computation 8:10* (1998) 1-31.

[S+98] Sernadas, A., Sernadas, C., Caleiro, C., Mossakowski, T.: Categorical Fibring of Logics with Terms and Binding Operators. In Gabbay, D., van Rijke, M. (eds.): *Frontiers of Combining Systems*. Research Studies Press (to appear).

[Tap97] Tapken, J.: Interactive and Compilative Simulation of PLC-Automata. In Hahn, W., Lehmann, A. (eds.): *Simulation in Industry, ESS`97*. Society for Computer Simulation (1997) 552-556.

[Tap98] Tapken, J.: MOBY/PLC – A Design Tool for Hierarchical Real-Time Automata. In: Astesiano, E. (ed.): *Fundamental Approaches to Software Engineering, FASE'98,* at ETAPS'98, Lisbon. LNCS 1382, Springer (1998) 326-329.

[TD98] Tapken, J., Dierks, H.: Moby/PLC – Graphical Development of PLC-Automata. In Ravn, A.P., Rischel, H. (eds.): *FTRTFT`98*, LNCS 1486, Springer (1998) 311-314.

[Tar96] Tarlecki, A: Moving between logical systems. In M. Haveraaen, O. Owe, O.-J. Dahl, eds.: *Recent Trends in Data Type Specifications, LNCS 1130*, 478-502. Springer, 1996.

[Tej99] Tej, H. (1999): HOL-CSP: Mechanised Formal Development of Concurrent Processes. Dissertation. (forthcoming)

[TW97] Tej, H., Wolff, B.: A Corrected Failure-Divergence Model for CSP in Isabelle / HOL. *Formal Methods Europe, FME'97*. LNCS 1313, Springer (1997) 318-337.

[UKP98] Urban, G., Kolinowitz, H.-J., Peleska, J.: A Survivable Avionics System for Space Applications. in *Proc. FTCS-28, 28th Annual Symposium on Fault-Tolerant Computing*, Munich, Germany, 1998.

[VMOD] V-Model: *Development Standard for IT Systems of the Federal Republic of Germany.* General Directives: 250: Process Lifecycle; 251: Methods Allocation; 252: Functional Tool Requirements. (1997).

[ZHR92] Zhou, C., Hoare, C.A.R., Ravn, A.P.: A Calculus of Durations. *Information Processing Letters 40(5)* (1992) 269-276.