Contents

Preface

Participants meeting at Dagstuhl
Considered the weather as too cool!
The idea of a pool
Would attract just a fool;
A firm rule is: wear things of good wool!
Bernd Krieg-Brückner

During the last 20 years several different formal and semi-formal specification techniques have been successfully developed and used. Applications comprise the specification of simple programs, data types and processes as well as complex hardware and software systems. The variety of specification techniques ranges from formal set theoretical, algebraic, and logic approaches for specifying sequential systems and from Petri-nets, process algebras, automata, and graph grammars for specifying concurrent and distributed behaviors to semi-formal software engineering methods for developing complex systems.

Formal and semi-formal approaches have their advantages and disadvantages: the informal diagrammatic methods are easier to understand and to apply but they can be ambiguous. Due to the different nature of the employed diagrams and descriptions it is often difficult to get a comprehensive view of all functional and dynamic properties. On the other hand, the formal approaches are more difficult to learn and require mathematical training. But they provide mathematical rigor for analysis and prototyping of designs. Verification is possible only with formal techniques.

Since a few years many researchers and research groups are putting more and more effort in closing this gap by integrating semi-formal and formal specification techniques. Their studies and experiences show the added value of combining semi-formal and formal techniques and at the same time open a whole range of new problems and questions which cannot be asked when studying formalisms in isolation.

In this seminar more than 40 scientists came together in 28 talks and two panel discussions to study possibilities and solutions for integrating and validating different formal and semi-formal specification techniques. Similarities and differences of formal and semi-formal specification formalisms as well as possibilities for combining such techniques were discussed. Most talks of this seminar analysed, compared, or integrated at least two such methods.

On behalf of all participants the organizers would like to thank the staff of Schloß Dagstuhl for providing an excellent environment to the conference. The support of the TMR programme of the European Community is gratefully acknowledged. Due to this programme it was possible to fund the participation of several young researchers and of three key note speakers.

The Organizers

Hartmut Ehrig *Gregor Engels* *Fernando Orejas* *Martin Wirsing*

Peace++: A Modal Logic-based Language for Specifying Cooperating Software Processes

Ilham Alloui

Université de Savoie, Annecy

Both semi-formal and formal languages are needed for respectively analyzing, specifying, enacting and evolving cooperating software processes (i.e. software engineering processes, workflow processes, etc.). To analyze processes, we propose a graphical language that allows to capture organisational, functional as well as dynamic aspects of a process. The resulting specification is to be validated by the user before going further into details as in the specification phase. For this latter, a second language which is formal is proposed in order to model cooperation (communication, coordination, goal sharing, etc.), based on a modal logic with operators for expressing goals, belief, time (past and future actions), possibility and necessity. The logic used is itself a combination of several logics into the so called "theory of rational action" of Cohen and Levesque. The aim on behalf this is to give an intentional semantics to cooperation of processes. The language proposes cooperation constructs founded on the concept of "communication act", i.e., sending or receiving a message during an interaction is controlled by a precondition, motivated by an intention and has an effect on the process (a postcondition). As software processes need not only to be analyzed and specified but also to be enacted and evolved, we adopted an object oriented reification technique for both process models and process instances with their enactment state. This concrete representation is executable and evolvable "on the fly". The conclusion is that in the software process domain, we need to use both semi-formal and formal specification techniques according to the process life-cycle.

Feature Interaction: Prevention or Detection?

Egidio Astesiano

Universitá di Genova

The general problem we address is "evolution in the sw development process", which is recognized to be challenging and of major importance nowadays. Here we deal with a particular aspect of that problem, namely the so-called "feature interaction" problem. That problem arose historically in the area of telephone systems, where it is typical to add a new user facilty (like the automatic repetition of busy numbers, the warning for another current call, etc.); at a recent meeting of the IFIP WG 2.2, G. Holzmann, of AT&T, reported that the number of such facilities may exceed 700. In simple words, a feature is a unit of update within a system (process); the problem is that the update could result in some perturbation of the system beyond the intention. A lot of related work has been done in very recent years; there is even an annual specially devoted international workshop. The dominant approach is aimed at the detection of feature interactions,namely at providing techniques for discovering bad interactions, usually via some model checking and/or verification tools. We see the value, but also the limits of such "a posteriori" approach and advocate a more preventive approach (slogan: prevention first), best expressed by Pamela Zave who calls for "an approach based on modular specifications and separation of concerns ... (allowing) to organize the specification so that it is easy to add without destroying its structure or desirable properties".

The approach we present is at two metalevels: methodological and technical. Indeed we propose a number of ideas, coherently embedded in a method, that can be reified in various formalisms. But we illustrate concretely the methodological aspects by means of a technical formalism, which has its own technical peculiarities. The formalism is based on on ideas developed by the authors in the years and now embedded in the CoFI/CASL method.

In order to separate the concerns and modularize the development, we distinguish some steps.

1. *Single feature level.* Here a feature is considered as an isolatad entity;

in our formalism it is modelled by a generalized labelled transition system and presented as a pre-feature specification (first-order conditional partial specification). Two significant technical aspects are: the states as collections of attribute values (a typical OO assumption) and the new concept of “transition type”, which will allow later to put together partial activities of different features related to the same event.

2. *Putting pre-features together.* We give a notion of pre-feature composition $F_1 + F_2$, which is partial, associative and commutative.

3. *Defining interactions.* Comparing $F_1 + F_2$ with, say, F_1, allows us to define ”interaction” as a discrepancy of the composition wrt F_1. It can be shown that a variety of kinds of interaction are definable, eg atomic or behavioural and each one with a wealth of different interesting variations. Note that interactions are in principle neutral, ie neither good nor bad, as it can be shown; thus we have to single out which are really unwanted.

4. *The concept of feature specification.* The complete notion of feature specification comes out as a pair: pre-feature + interaction requirements where the requirements are constraints about the interaction with other features, in some appropriate logic. A feature specification has two kinds of semantics: basic and complete. While basic semantics is just the generalized labelled transition system obtained by logical deduction, complete semantics is a class of systems which satisfy the specification under an *antiframe assumption*: roughly, what is not forbidden may happen (ie variation of attributes and moves). The complete semantics gives a basic insight of what could really happen when adding other features and the interaction requirements should constrain the complete semantics to exhibit only sensible behaviours.

5. *Handling interactions.* In order to guide the specification of interaction requirements, we introduce the concept of ”discipline” for kinds of interactions: a kind of interaction can be disciplined by some logical formulae whenever those formulae are able to prevent interactions of that kind. For example, what we call atomic interactions can be disciplined by some appropriate safety and liveness onditions. Thus, on the basis of complete semantics, we are guided to prevent unwanted interactions by expressing appropriate interaction requirements. When

trying to compose two features, the composition is defined only if the interaction requirements of both are satisfied; here is where the techniques of verification and model checking come at hand, but only after a careful modularization and prevention.

As a last remark, notice that our approach is compositional, since the composition of two (pre-) fatures is again a (pre-) feature. Please contact the authors for copies of related papers and/or criticism/enquiries at the following addresses {`reggio,astes`}`@disi.unige.it`

On the integration of formal and semi-formal techniques using ASMs

Egon Börger

Universitá di Pisa

We explain how Abstract State Machines (ASMs) allow one to integrate so called formal and semi-formal techniques into a practical framework for the development of complex systems.

ASMs offer possibilities for both horizontal and vertical integration. On the one side, through the ASM classification of functions (into basic and derived, static and dynamic and into monitored, controlled and shared functions) different methods to specify parts of the system can be incorporated at any point into the rigorous description of the overall system. On the other side different ASMs can be linked to a hierarchy of stepwise refined models for the specification, design and analysis phase of software development. In both cases the fundamental mechanism is the determination of the appropriate level of abstraction for the system description. A special case of this fine tuning is the distinction of so called formal and semi-formal description techniques.

A particularly important example of this formal versus semi-formal description problem is offered by the ground model problem of software engineering, namely the problem to define in a sufficiently rigorous way the informally given requirements of a software system to be designed. We show that the most general abstraction principle which is built into ASMs allows one to solve this problem in a satisfactory way, namely by building models which are formulated in the application domain language and which can be inspected (and possibly falsified) by the customer (who may have no software or computer expertise).

We illustrate our arguments by two sorts of example for details of which we point to the literature: the definition of a programming language and of its (provably correct) implementation on a (virtual) machine - the case of the ISO Prolog standard,of Occam and of Java and of their implementation on the WAM, the Transputer and the Java Virtual Machine respectively — and the development of control software starting from appropriate ground models for the informally given requirements — the steam boiler and the production cell case studies.

References

[1] E. Boerger: Why use of evolving algebras for hardware and software engineering. in: M. Bartosek, J. Staudek, J. Wiedermann (Eds), SOFSEM'95 22nd Seminar on Current Trends in Theory and Practice of Informatics. Springer Lecture Notes In Computer Science, vol. 1012, 1995, pp.236–271.

[2] E. Boerger and D. Rosenzweig: A Mathematical Definition of Full Prolog. in: *Science of Computer Programming* 24 (1995) 249–286.

[3] E. Boerger and D. Rosenzweig: The WAM - Definition and Compiler Correctness. In: *Logic Programming: Formal Methods and Practical Applications* (C. Beierle, L. Plümer, Eds.), Elsevier Science B.V./North-Holland, Series in Computer Science and Artificial Intelligence, 1995, pp. 20–90 (chapter 2).

[4] E. Boerger and I. Durdanovic: Correctness of Compiling Occam to Transputer Code. in: The Computer Journal, Vol. 39, No.1, pp.52-92, 1996.

[5] C. Beierle, E. Boerger, I. Durdanovic, U. Glaesser, E. Riccobene Refining abstract machine specifications of the steam boiler control to well documented executable code. in: J.-R. Abrial, E.Boerger, H. Langmaack (Eds.): Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control. Springer LNCS State–of–the–Art Survey, vol. 1165, 1996, 52-78.

[6] E. Boerger and L. Mearelli: Integrating ASMs into the Software Development Life Cycle. in: Journal of Universal Computer Science, Special ASM Issue, 3.5 (1997), 603-665.

[7] E. Boerger and W. Schulte: Programmer friendly modular definition of the semantics of Java. in: J. Alves-Foss (Ed.): Formal Syntax and Semantics of Java, Springer LNCS 1998 (to appear).

[8] E. Boerger and W. Schulte: Defining the JavaVirtual Machine as Platform for Provably Correct Java Compilation. in: J. Gruska, J. Zlatuska (Eds.): Proc. MFCS'98. Springer LNCS 1998 (to appear)

Concurrency and Data Types: A Specification Method. An Example with LOTOS

Christine Choppy

Université de Nantes
(Joint work with Pascal Poizat and Jean-Claude Royer, Université de Nantes)

This work is motivated by the fact that methods are needed to help using formal specifications in a practical way. It aims at providing some help in establishing specifications that involve both concurrency and data types, and it is here developed for LOTOS specifications (but it could be adapted to other formalisms).

Among the few existing methods for LOTOS specification, some follow the "constraint-oriented" approach (where the emphasis is put on identifying components that can run in parallel) and others follow the "state-oriented" approach (where processes are modelled using automata and data types are associated to processes).

Our method is using both approaches: the constraint-oriented one is used to decompose processes into sequential sub-processes, and the state-oriented one is used for the sequential sub-processes specification.

Our method adds:

- guidelines for how to apply these in a given order,
- a systematic and semi-automatic construction of the sequential components automata,
- the automatic derivation of the LOTOS behaviour specification from the automata, and
- assistance for the algebraic specification of the static part provided through the automatic processing of the information gathered in the previous steps.

This method is illustrated through a simple example, a hospital.

The Formalisation of SOCCA using Z

J. H. M. Dassen

Universiteit Leiden

The semi-formal object-oriented visual specification language part of SOCCA[1] is described in a high-level fashion. Its eclecticism (careful composition of

[1] `http://www.wi.LeidenUniv.nl/CS/SEIS/socca.html`

selected other techniques (class diagrams, State Transition Diagrams and the subprocess and trap extensions thereof that facilitate description of communication)) is illustrated and is advocated as a way to successfully integrate (semi-)formal specification languages and notations.

The primary goal of the ongoing work to formalise SOCCA using Z is discussed: improving understanding of how object orientation and precise descriptions of communication can be combined.

The approach is compared to one based on developing a meta-class diagram.

Three examples of the benefits the process of formalisation is bringing (clearly identifying the commonality between attributes and methods; the necessity of addressing the inheritance of relationships; three conceptual levels in describing SOCCA) are briefly discussed.

It is argued that the process of formalisation is perhaps as important as its product.

Petri Nets as an Interlingua for Semi-formal and Formal Specification and Analysis

Jörg Desel

Universität Karlsruhe

There is an apparent gap between existing semi-formal techniques for specification and modeling of processes (application layer) and precise workflow and process languages (implementation layer). Petri nets constitute an appropriate candidate for a linking model on the logical layer. Arguments include:

- mathematical foundation,
- expressive power,

- possible integration with data / roles / organization / . . . ,
- analysis and verification techniques,
- existing tools,
- Petri net based techniques frequently used on the application layer,
- Petri net based techniques often suggested for the implementation layer.

Vicinity respecting net morphisms support design and transformation of net models and respect elementary relations between net elements. Token transformation techniques relate different high-level net models that support analysis, simulation, and compilation, respectively.

Finally, concepts developped in the project VIP (supported by DFG) are presented:

- marked Petri nets as a specification of possible concurrent runs (processes),
- a graphical language for the formulation of requirements,
- generation of processes,
- validation via visualization of processes and browsing options,
- efficient analysis techniques of processes w.r.t. requirements, employing the representation as occurrence nets and graph algorithms.

Compiler Support for the Specification and Verification of Software Systems

Klaus Diderich

Technische Universität Berlin

We propose to extend the compiler of a programming language with support for specification and verification. This integrative approach allows to keep related information together during software engineering. Verification is supported by the inclusion of proofs which can be checked by the compiler. Such an environment allows the programmer to use formal and semi-formal specifications and verifications as is appropriate. Several styles of software development are supported: quick-and-dirty programming (for upwards compatibility), formal treatment of "hot spots", and finally full-fledged formal development. Development of secure programs is supported by the possibility of an independent proof check. Algebraic context conditions which usually must be ensured by the user, can now be checked by the compiler.

Some of the problem areas in this approach are language design, handling of (semi-)formal specifications and handling of (semi-)formal verifications. The language designer has to reconcile specification, programming and verification aspects. Inclusion of semi-formal specifications is only possible for linear textual representations and must be supported by appropriate justification methods. Different verification and justification methods are related to different stages in the compilation process which makes inclusion complicated.

Modelling of Collaboration with UML and SOCCA

Gregor Engels

Universität Paderborn

(Joint work with Luuk Groenewegen, Rijksuniversiteit Leiden)

In contrast to the well-understood techniques for modelling structured aspects, the modelling of dynamic aspects is still a weak point in current object-oriented modelling approaches. In particular, generally accepted techniques for the modelling of coordinated colloboration within a society of objects do not exist yet. The talk discusses and compares two exciting approaches for collaboration modelling. These are the so-called collaboration diagrams of the OMG standardized language UML and the object-oriented specification language SOCCA, which have been developped during the last 5 years at Leiden University. The comparison is based on a classification of interaction patterns, where a distinction is made between synchronous, asynchronous, future synchronous and restricted asynchronous behaviour at the start and the end of an interaction between a sender and a receiver. The two main results of the discussion and comparison are as follows. First, due to a missing clear semantics of collaboration diagrams, the comparison has to be based on assumptions have to interpret UML notations. Second, the encoding of synchronisation constraints in textual constraints on execution orders is less expressive and understandable than the graphical representation of synchronisation constraints in the language SOCCA. In addition, the only usage of state transition diagram notation for the dynamic model of classes, the functional models of operations and the synchronisation of operations supports a uniform, but modular specification approach.

Temporal Ordering of Actions vs. Rule Based Specifications (An Application of Algebra Transformation Systems)

Martin Grosse-Rhode

Technische Universität Berlin

Temporal ordering of actions and rule based state changes are two fundamental approaches to the specification of reactive systems. Algebra transformation systems are two layered formal models that can be used for a formal comparison of specifications that belong to the different approaches. This comparison allows e.g. consistency checks in multiple viewpoint modelling, where the specification of different aspects of a system in different notations or different types of formal models is supported. In this talk I persent a translation of two specifications of the alternating bit protocol to algebra transformation systems. The first one is a CCS specification from Milner's book, exemplifying the temporal ordering of actions approach, the second one is a UNITY specification (program) taken from the book of Chandy and Misra as a representative of the rule based approach. Then different kinds of operations on algebra transformation systems can be used to compare the specifications via the translations. It turns out that the specifications show relevant differences, due to the different approaches. In particular, using shared variables as communication means instead of message passing supports a different distribution of the behaviour to the components of the protocol. On the other hand the translation yields some "added value", such as making explicit different roles of features as e.g. input/output in CCS, the possiblity to make explicit data states in process specifications, and, vice versa, adding control flow information to rule based specifications.

From Informal Requirements to Formal Specifications: A Systematic Transition

Maritta Heisel

Universität Magdeburg
(Joint work with Jeanine Souquieres, LORIA and Université de Nancy 2)

We propose a method for the elicitation and the expression of requirements. The requirements can then be transformed in a systematic way into a formal specification that is a suitable basis for design and implementation of a software system. The approach — which distinguishes between requirements and specifications — gives methodological support for requirements elicitation and specification development. It does not introduce a new language but builds on known techniques.

Observational Logic

Rolf Hennicker

Universität München
(Joint work with Michel Bidoit, CNRS and
Ecole Normale Supérieur de Cachan)

We propose a uniform logical framework for specifying observational properties of state based systems, in particular of object-oriented programs.

Formally we introduce the institution of "'observational logic"' which is based on the idea that an observational signature contains a distinguished set of "'observers"' which determine an indistinguishability relation (called "'observational equality"') between the elements of an algebra.

Based on the institution of observational logic we define structured observational specifications by using the institution independent specification building operations of Sannella and Tarlecki.

For proving (first-order) observational properties of specifications we first construct a sound and complete proof system for observational logic. Then we use this proof system to obtain a sound and complete proof system for structured observational specifications by applying a general institution independent result of Borzyszkowski.

Towards an Integration of UML/OCL and Algebraic Specification

Heinrich Hußmann

Technische Universität Dresden

This talk claims that algebraic specification technology can provide significant contributions to the further development of the standard software engineering notation UML and its formal extension OCL. It is argued that a major future challenge will come from a seamless integration between semiformal software engineering notations and enhanced formal notations. This integration has to be achieved in such a way that it becomes easy to switch towards a more formal specification only for those parts and at those stages of a development project where this is required.

One important precondition for such an integration effort has been fulfilled now by the standardisation of the object-oriented modelling notation UML. UML is more precisely defined than most of its predecessors and seems to gain acceptance in industry rapidly. As a formal extension of UML (and officially a part of it), the Object Constraint Language (OCL) has been introduced. According to the claims made by its inventors, OCL can be used to write unambiguous specifications like a formal specification language, but is more usable by the 'average business or system modeler', since it does not require mathematical background.

In order to investigate this claim, an alternative notation for object constraints is suggested which is called Algebraic Object Constraints (AOC). AOC is based on the language and calculus of predicate (and temporal) logic but is used for the enhancement of UML specifications by precise annotations.

A comparison of both notations shows that AOC is at least as easy to comprehend as OCL. Moreover, AOC enables the formulation of object-independent invariants (but also, of course, object-local constraints). It is easier to apply deductive tools to AOC since there are already powerful calculi and tools available.

The conclusion is that UML and OCL are to be taken serious by the formal methods community, and can be seen as a chance for a transition towards 'industry-proof' formal development support. OCL is particularly interesting, since it claims to be superior to pure formal methods, but there is no clear evidence for this advantage. So more reserach is needed for alternative complementary formal notations to UML, for experiments on usability and teachability, and for experiments in the application of deductive tools.

Towards an Integration of Message Sequence Charts and Timed Maude

Piotr Kosiuczenko

Universität München

(Joint work with Martin Wirsing, Universität München)

The topic of the talk was an integration of a graphical and a formal method and building a unifying specification formalism which can support different software views: functional, data, process, and time; in particular integration of Timed Maude and Message Sequence Charts (MSC). Maude is a formal object-oriented specification language which combines algebraic specification techniques for describing complex data structures with term rewriting to deal with dynamic behaviour. MSC is a graphical trace language for describing and specifying the communication behaviour of distributed systems by means of message interchange. We showed that MSC and Timed Maude fit well together: on one hand, we expanded MSC-96 with primitives like multicast, synchronous communication, and multicast, which are available in Maude. Those new features will probably appear in MSC-2000. On the other hand, MSC provided high-level composition mechanisms and a graphical notation for Maude. We expanded Timed Simple Maude by two composition operators for sequential and parallel composition. Both operators are formalized using a syntactic substitution operator which is derived from a pushout construction.

Synchronization Constraints in Object Interfaces

Bernd J. Krämer

Fernuniversität Hagen

In distributed computing environments no assumptions can be made about the order in which the operations visible at a server object interface are invoked. In critical applications, unsynchronized accesses to shared resources, illegal execution orders or violations of capacity constraints need to be detected and ruled out to guarantee the consistency of a server's state. However, interface definition languages (IDLs) of current middleware platforms such as CORBA, DCE or DCOM - which play an increasing role to distributed application developers - offer no means to formally document synchronisation constraints or other types of qualitative requirements. They merely capture type and operation signatures.

Based on a partial order model of computation, we propose a concise notation for specifying synchronization constraints at object interfaces. To ensure compatibility with the CORBA standard, these annotations are included as comments in IDL interfaces. They are ignored by standard IDL compilers, while a separate compilation framework generates C++ code from a given set of synchronization constraints. This code provides special synchronization methods that can be wrapped around the application developer's operation implementation to guard the execution of operation invocations depending on the object's invocation history. Different solutions how the synchronization code can be transparently combined with the skeleton code produced by the IDL compiler at hand and the developer's method implementation classes are sketched. We are currently investigating the potential of these soluations to act as design patterns for specification supplements defining operation semantics in terms of pre-/post-conditions, timing constraints or other quality properties.

Combination of Methods and Tools in the UniForM Workbench

Bernd Krieg-Brückner

Universität Bremen

Formal notions of translation, embedding, combination and projection are supported by logic representation in Isabelle/HOL. So far, Z and its Mathematical Toolkit, CSP and its Process Algebra have been encoded and proved correct; similarly, the static semantics of CASL in-the-small has been represented including overload resolution. Combinations HOL+CSP, CASL+CSP, Z+CSP, and extension for real-time are presently being realised; projection to component languages or sublanguages shall be supported by decomposition transformations.

Tools communicate in the UniForM Workbench as a network of concurrent agents. Control and data integration are achieved by the Subsystem Interaction Manager and Repository Manager, presentation integration by the User Interaction Manager (with interfaces to the graph visualisation system daVinci, Tcl/Tk, etc.), comprising a total of 45k lines of Haskell. Haskell is the internal integration language, providing a high degree of abstraction, structuring and type-safety. External tools are wrapped into a Haskell interface of adaptors etc.; we plan an adaptation of CORBA/IDL.

Integration into the development process is under way by tools supporting the V-model and its tailoring to a company or a particular project. The V-model will be adapted to Formal Methods; we plan to provide a service for integrating foreign methods and tools with the UniForM Workbench. Development steps are formalised by transformation rules; these can be proved correct and applied in a graphical, gesture-oriented way with the verification system IsaWin and the transformation application system TAS, both based on Isabelle. Development scripts can be replayed and abstracted to new rules; thus reusability of the development process becomes possible.

A Case Study on Self-Stabilization Revisited or: A Case Made for Visual Proofs in System Verification

Stephan Merz

Universität München

We review a case study described by Qadeer and Shankar [2] who use PVS to verify a self-stabilizing algorithm introduced by Dijkstra [1] in 1974. Their formalization is based on a mathematical proof that performs an elaborate sequence of steps including induction over natural numbers, well-founded induction, and temporal-logic style arguments such as proofs of invariance and liveness conditions. The formalization as well as the sequence of PVS interactions was considered nontrivial by the authors.

We contrast this proof with a direct termination proof based on a valuation function over a well-founded domain. There, it suffices to apply one single, standard rule, but the resulting proof conditions are quite daunting, and the proof does not shed much light on the structure of the algorithm. We argue that the best way to present the proof is in the form of a transition system whose nodes are labelled with predicates that describe abstract system states, and whose transitions represent a superset of the actual system transitions. In order to capture liveness conditions, we allow the user to specify valuation functions over well-founded domains and to annotate the edges in order to indicate steps that (strongly or weakly) decrease these valuation functions. The diagram represents the high-level structure of the proof and gives rise to two kinds of verification conditions:

- non-temporal conditions on state predicates and individual transitions, which can be verified with the help of an interactive proof assistant, and
- an abstract interpretation of the diagram as an automaton (enhanced by liveness conditions generated from the edge annotations) that can be analyzed using automatic tools such as a model checker.

For this specific example, we have manually extracted the corresponding verification conditions and have discharged them using Isabelle/HOL and a PTL decision procedure, respectively. The resulting proof script appears to be considerably shorter and simpler than the PVS formalization of [2]. We strongly believe that this technique applies widely in the area of system verification and suggest to build a visual front-end that serves to integrate automatic and interactive proof methods.

References

[1] E. W. Dijkstra: Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[2] S. Qadeer and N. Shankar: Verifying a Self-Stabilizing Mutual Exclusion Algorithm. R. Cleaveland and S. Smolka (eds): Proceedings of the IFIP Working Conference on Programming Concepts and Methods. Chapman and Hall, June 1998.

Integration and Classification of Data Type and Process Specification Techniques

Fernando Orejas

Universitat Politècnica de Catalunya
(Joint work with Hartmut Ehrig, Technische Universität Berlin)

The integration of different kinds of data type and process specification techniques has become an important issue in system specification. Based on several well-known examples of integrated specification techniques, we propose an integration paradigm for system specification which provides a unified approach on a conceptual level. The key idea is to consider four different layers which correspond to different kinds of integrated views of system specification. This integration paradigm can be considered as an extension of the concept of dynamic abstract data types which was proposed by the same authors in 1994. A large variety of data type and process-driven specification approaches are briefly discussed as instances of the integration paradigm leading to a classification of these approaches.

Panel Discussion: Integration of Formal Specification Methods

Fernando Orejas

Michel Bidoit, Friedrich von Henke,
Bernd Krieg-Brückner, Roel Wieringa

The aim of this panel was to discuss the need and the adequacy to integrate different specification approaches correponding to different views of a system description. Two opposite opinions were presented in the panel. On the one hand, the fact that current specification approaches mainly address some specific aspect or view of complex systems, was considered to be a reason for the need to study the combination of complementary methods. On the other hand, it was considered that a formalism integrating very different specification approaches would surely be a “monster” too complex to use and to master.

Panel Discussion: Formalization of UML!? A Challenge for Theoreticians as well as Practitioners in Computer Science?

Gregor Engels

Gianna Reggio, Bernhard Rumpe,
Gunter Saake, Wilhelm Schäfer

The Unified Modelling Language (UML) has been accepted by the OMG (Object Management Group) as standard formalism for object-oriented software models. While this standardization effort has been accepted as a very positive development by most of the people in the oo community, a lot of discussion arose about the quality of the available UML documentation. In particular, the description on the semantics of UML caused more questions than answers. A closer look at this document shows that at the moment only an informal, sometimes even inconsistent description of the semantics exists. Thus, the following questions should be discussed:

1. Is UML an informal or semi-formal language?
2. Should UML become a formal language?
3. Is there a chance to formalize UML?
4. Which approach is appropriate to define UML's semantics?

The four panelists expressed their opinion about these and related questions, which stimulated the audience to participate in an interesting discussion.

Gunter Saake pointed out that a formalization of UML is mainly interesting for the developers of UML and supporting tools, but not for the typical user of the notations. Users are much more interested in a methodological support and clear guidelines how to deploy UML than in a mathematical formalization. Besides that, Gunter gave some provocative arguments, why formalization attempts will fail. For instance, he claimed that UML is too complex for a single formalism, and the fact that extension is part of the standard makes any precise formalization impossible.

Gianna Reggio was less pessimistic and proposed a logic-based approach for formalizing UML. She pointed out that it is important to understand the interdependencies between the different diagram types. This is a prerequisite for a consistent mapping of diagrams into an integrating common model. In addition, she proposed some extensions to UML diagrams as, for instance, more precise inscriptions on the diagrams.

Wilhelm Schäfer pointed out that he does not believe in the bing-bang approach of UML. He is not convinced that such a general language as UML will be successful in industrial applications. He proposed to work on domain-specific adjustments of UML. In his view, the current version of UML is much too complex and general and cannot be viewed as a standard due to the inconsistent and superficial explanation of the language constructs.

Bernhard Rumpe concentrated on a specific part of UML, the Object Constraint Language (OCL), to express constraints and integrity conditions in UML diagrams. He gave a concrete example and made clear that the current syntax proposal of OCL is not appropriate, and on the other side a precise semantics is still missing. He gave this as an example for the overall imprecise and inconsistent description of syntax and semantics of UML.

The discussion by the audience showed that most of the participants agreed that UML is an important development and a great challenge for practitioners as well as for theoreticians. The discussion has to be continued which formal approach is appropriate to define the semantics (of parts) of UML. On the other side, pragmatic guidelines and methods have to be discussed how to use (parts of) UML in software development projects.

Formal Semantics of UML State Diagrams Using Graph Transformations

Francesco Parisi-Presicce

Universitá Roma La Sapienza
(Joint work with Martin Gogolla, Universität Bremen)

As an intermediate step between UML diagrams and a general comprehensive semantical framework, state diagrams are transformed into labelled graphs, very similar to the original diagrams but forcing an unambiguous interpretation. Various semantical frameworks (such as streams, temporal logic, graph transformation systems) are applicable to the resulting graphs (the semantics of UML state diagrams).

The expansion of nested state diagrams is accomplished via very simple rules in the double pushout approach to graph transformations by:

1. adding boundary nodes introducing a precise interface for the state to be expanded
2. replacing the state with its internal structure (expansion)
3. removing the boundary nodes.

The first and last rules are "methodological" while the second one is "application dependent" and provided by the designer of the nested state. Similar sets of rules are used for stubbed transitions and for exit nodes. The approach gives an intuitive way of achieving a normal form for nested state diagrams by means of graph transformations and can be used for other forms of UML diagrams, such as class diagrams and sequence diagrams.

References

[1] M. Gogolla and F. Parisi-Presicce: State Diagrams in UML: a Formal Semantics using Graphs Transformations. Techn.Rep. 97/15, Dip. Scienze dell'Informazione, Univ. Roma La Sapienza, Dec. 1997. (reduced version in Proc. PSMT'98 Workshop, TUM-I9803, Apr. 1998)

Montages — Towards a Popular Semantics

Alfonso Pierantonio

Universitá di L'Aquila

Over the last years, a number of formalisms have been proposed. They have been successfully used to investigate fragments of languages and to discover flaws (in some formal sense) in the design of languages. Unfortunately, language designers make still little use of formal methods and most of the languages are specified informally. Although Scott-Stratchey-style denotational semantics introduced 25 years ago the revolutionary idea to extend BNF to semantics, the mission to provide a semantical analogous to BNF was never accomplished. It is surprising and disappointing that language design is nowadays performed by using methods of the 1970s and 1980s. In other words, there was not the necessary technology transfer from semantics research to programming language development.

On the other hand, semantics research followed the trend towards higher levels of specialization, at the expense of computational clarity. As carefully pointed out in [8], there is a plethoric spread of techniques and formalisms which generated more experts than general users.

General users would like to use BNF-like techniques to comprehend the meaning of programs. This is due to the quality factors of syntactical formalisms, such as EBNF, i.e. to the fact that syntax descriptions are writable, modifiable, reusable and readable. If semantics descriptions had such qualities as well, then they could be used for language manuals, to record design decisions which in turn could be conveyed to language implementors, and finally they could be analyzed and entered in rapid prototyping tools.

These heavy demands advocate the need for a popular semantics, i.e. a framework which has desirable pragmatic qualities that make it an excellent tool for the language designer. Unfortunately these qualities are not possessed by the mostly known semantics formalisms such as natural semantics and denotational semantics [10].

Having as major aim the development of a popular semantics framework, we designed Montages [5] and its tool companion Gem-Mex [1, 2]. Montages constitute a specification formalism useful for describing all aspects of programming languages. Syntax, static analysis and semantics, and dynamic semantics are given in an unambiguous and coherent way by means of semi–visual descriptions. The static aspects of Montages resemble control and data flow graphs, and the overall specifications are similar in structure, length, and complexity to those found in common language manuals. Montages are a theoretical basis and tool for a number of activities from initial language design to language implementation. Experience with specifications of Oberon [6], Java [9], and SQL (ISO9075) [3] have shown that existing languages can be easily described and documented. In [7] we report on a domain specific language that has been designed and prototype from scratch using Montages in an industrial context.

The mathematical semantics of Montages is given by means of Abstract State Machines (formally called Evolving Algebras) [4]. In short, ASMs are a state–based formalism in which a state is updated in discrete time steps. Unlike most state based systems, the state is given by an algebra, that is, a collection of functions and universes. The state transitions are given by rules that update functions pointwise and extend universes with new elements.

Montages engineered the ASM's approach to programming language semantics showing how to model consistently not only the dynamic semantics, but the static analysis and semantics as well. In particular, we describe how to define intensionally the abstract syntax, i.e. the control and data flow, starting from the concrete one. This mapping between concrete and abstract syntax is provided by means of graphs which confer to the specification a great intelligibility.

A language specification is given by a collection of Montages, which is hierarchically structured according to the rules of the corresponding context-free grammar given in EBNF. Each Montage is a "BNF-extension-to-semantics", that is a self contained description in which all the properties of a given construct are formally defined.

Such Montage specifications can be fed to the Gem-Mex [1, 2] tool. It is a complex system which assists the designer in editing and composing specifications, generating documentation and language implementations, respectively. It consists of a number of interconnected components

- the Graphical Editor for Montages (Gem) is a sophisticated graphical editor in which Montages can be entered; furthermore documentation can be generated automatically;

- the Montages executable generator (Mex) which automatically generates correct and efficient implementations of the language;
- the generic animation and debugger tool visualizes the static and dynamic behavior of the specified language at a symbolic level; source programs written in the specified language can be animated and inspected in a visual environment.

The whole development of a programming language can be supported with an effective impact on the productivity and robustness of the design. The designer can enter the specification, browse it and especially maintain it. Specifications may evolve in time and modifications can be localized within very neat boundaries. By doing so, different experimentation can take place with different versions of the syntax and semantics of the specified language in a very short time.

References

[1] M. Anlauff, P. W. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.

[2] M. Anlauff, P. W. Kutter, and A. Pierantonio. The Gem-Mex Tool Homepage. `http://www.first.gmd.de/~ma/gem/`, 1997.

[3] B. DiFranco. Specification of ISO SQL using Montages. Master's thesis, Universita di l'Aquila, 1997.

[4] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[5] P. W. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *JUCS, Springer*, 3(5):416–442, 1997.

[6] P. W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *JUCS, Springer*, 3(5):443–503, 1997.

[7] P. W. Kutter, D. Schweizer, and L. Thiele. Integrating Formal Domain-Specific Language Design in the Software Life Cycle. to appear in proceedings of Current Trends in Applied Formal Methods, October 1998, Boppard Germany.

[8] D. A. Schmidt. On the Need for a Popular Formal Semantics. *Sigplan Notices*, 32(1):115–116, 1997.

[9] C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, University of Michigan EECS Department Technical Report, 1997.

[10] D. A. Watt. Why don't Programming Language Designers use Formal Methods? In R. Barros, editor, *Proc. SEMISH'96*, pages 1–16, University of Pernambuco, Recife, Brazile, 1996.

CASL-CHARTS: An Initial Proposal for Integrating CASL and Statecharts

Gianna Reggio

Universitá di Genova

(Joint work with Lorenzo Repetto)

CASL-CHARTS is a graphic formalism for the specification of reactive systems defined by combining Statecharts [1] (the variant supported by Statemate) and CASL [2]. Statecharts are a well-known graphic language for reactive systems, while CASL is the common algebraic specification language developed within the CoFI initiative.

The idea of the combination is to use CASL for the data part and the graphic part of Statecharts for the reactivity aspects of a system. Thus a CASL-CHARTS specification will be a pair consisting of a CASL specification SP defining some data structures, which will be used by a Statechart CH.

The semantics will be given in two steps:

1. give the semantic to SP (an algebra D)
2. give the semantics to CH using D (as for classic charts).

The proposed formalism improves classic Statecharts, allowing

- more abstract specifications;
- more compact/readable specifications;
- simpler ways to present the charts and to give their semantics.

References

[1] D. Harel, Statecharts: A Visual Formalism for Complex Systems Science of Computer Programming, 1987, vol. 8.

[2] P. D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, Proc. TAPSOFT '97 number 1214 in Lecture Notes in Computer Science, pages 115–137, Berlin, 1997. Springer Verlag.

Graphic Formalisms

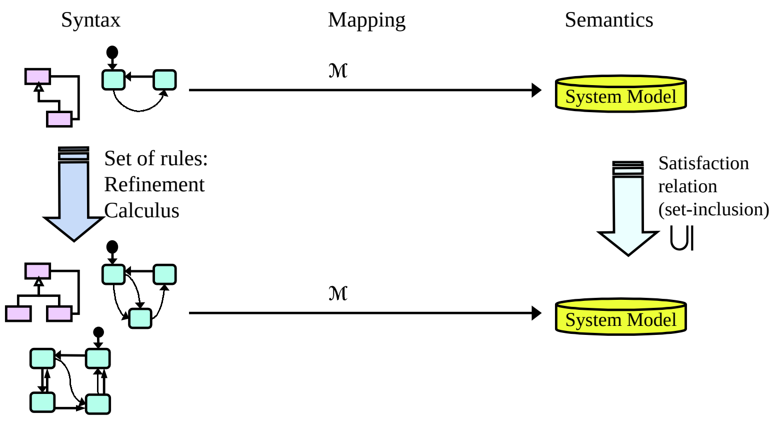
Bernhard Rumpe

Technische Universität München

Today there is a widespread belief that graphic notations are informal or at most semi-formal. This belief especially comes from the area of object-oriented software engineering methods, which today very much rely on UML. That it is so widespread is somewhat surprising, given the existence of quite a number of diagrammatic formalisms such as graph grammars, Petri nets, finite automata, etc.

In the talk we present work done together with Jan Philipps and other colleagues in Munich, in which we not only formalized two graphical notations — one for behavior automata and one for data flow diagrams —, but also developed a set of refinement rules for them. Both refinement calculi have been proven correct with respect to a semantics based on stream processing functions, such that the diagram shown below commutes.

It is argued that it is not the formalization itself that is of value (except perhaps for those few who do the formalization), but rather the results gained from it. Typical results are context conditions for each single notation and for the integration of several notations, as well as transformation rules for code generation, refinement and property derivation. We firmly believe that we must employ a considerable amount of these techniques for UML in the future to make it an even better success.



Evolving Specifications in Information Systems Design

Gunter Saake

Universität Magdeburg

Information system design differs from program construction as much as house building differs from maintaining a large city. Information systems are long-living, evolving software constructs integrating several, partial autonomous subsystems. This talk focusses on the evolution aspect of such systems and how evolution can be handled during a formalized modelling phase.

The key idea of integrating evolution aspects into classical information system specification is the separation of a fixed 'rigid' specification part from an evolving part containing specification fragments describing the adapted specification parts. The language Troll which is based on object orientation and temporal logic can be extended to describe evolving objects without inventing new formalisms. Currently, two temporal logic extensions are developped to build a logic framework for reasoning about such evolving specifications. These logics differ in the aspect how to interprete the state-dependent specification fragmenst: dyOSL [1] interpretes them at run-time, whereas the second development U2 uses a compilation to a temporal logic without reflection but explicit mutation states.

References

[1] S. Conrad and J. Ramos and G. Saake and C. Sernadas: Evolving Logical Specification in Information Systems. J. Chomicki and G. Saake (eds.): Logics for Databases and Information Systems. Kluwer Academic Publishers, Boston, 1998, chapter 7, pp. 199–228.

A Dedicated Software Process Design Language

Wilhelm Schäfer

Universität Paderborn
(Joint work with Gerald Junkermann, Sabine Sachweh and Stefan Wolf)

The work described here was motivated by the fact that commercially available software configuration management (CM) tools are difficult to adapt to different companies" needs, i.e. CM processes constitute a major part of the workflow of a software producing company and differ (sometimes significantly) between different companies. If at all, available tools only provide a Shellscript-like language to adjust a tool. This makes the usually complex CM-process definitions difficult to understand and maintain.

Our approach is to take an object-oriented specification language, namely a subset of UML (Unified Modeling Language). This subset has been formally defined in terms of its statics and dynamic semantics to enable domain specific analysis which includes to check properties like special types of cycles across different statecharts, deadlocks etc. Those properties indicate errors in the model. Similarly, simulation is used to find out errors. The dynamic semantics and according simulation is based on mapping the OO-specification language to PROLOG. Finally, the language includes a particular non-standard transaction protocol which supports to define synchronisation of concurrent accesses of users to the same version of a document. This protocol avoids the problems of so-called "long-term" transactions in the context of software processes.

A Formal Model for SDL Specifications based on Timed Rewriting Logic

L. J. Steggles

University of Newcastle

SDL (Specification and Description Language) is a standard industrial formal description technique for real–time distributed systems which is based on communicating finite state machines. Despite its wide spread use and industrial importance SDL lacks at present a complete and integrated formal semantics. In this talk we address this shortfall by presenting a formal semantics for SDL using a new algebraic formalism called Timed Rewriting Logic (TRL). TRL is a specification formalism which extends standard algebraic specification techniques by allowing the dynamic behaviour of systems to be axiomatised using term rewriting rules. The rewrite rules can be labelled with time constraints which provide a means of reasoning about time elapse in real–time systems. The formal semantics we develop captures in an intuitive way the hierarchical structure of SDL specifications and integrates within one formalism the static and dynamic aspects of an SDL system. It also provides a natural basis for analysing, verifying, testing and composing SDL systems. We demonstrate the approach we develop by considering modelling an SDL specification for the so called bump game.

Analysing Designs

Harald Störrle

Universität München

The functional evolution (read: maintenance) of large software systems has been a challenge for computer scientists ever since. Obviously, working on the code level has not provided sufficient results. Working on the design level, however, still looks promising. For work on this abstraction level one would need tool support. Among other things, the equivalent of a type-checker — one might call it a "consistency checker" or such like — is a very interesting tool.

In my talk I use the Unified Modelling Language (UML) to express designs syntactically, and high-level Petri-nets to represent them semantically. These choices are justified pragmatically. I give some examples of errors a designer might introduce inadvertently and briefly show how the formal semantics helps to discover these errors.

Moving Specifications Between Logical Systems

Andrzej Tarlecki

Uniwersytet Warszawski and Polska Akademia Nauk

In the process of systematic development of large software systems from their formal specifications, it is often most natural to use a number of different logical systems to specify various aspects of system behaviour, to describe various modules of the resulting composite heterogeneous system, and to capture features typical for different stages of system development. An initial step towards precise foundations for this is to formalize the concept of a logical system, and then to develop a formal framework for working within an arbitrary but fixed logical system. Much work in this direction has been done based on the theory of *institutions* (the concept introduced by Goguen and Burstall in the early 80s). Then, a necessary prerequisite for specifications/developments spanning a number of logical systems is some notion of a morphism between institutions to allow specifications and developments to migrate from one institution to another.

In this talk I present rudimentary notions to relate (model components of) institutions and show how these are sufficient for moving specifications between the institutions so related. Consequently, given a diagram of institutions related in such a way, one can build and use specifications that may exploit the logical power of the entire diagram, rather than of any single institution within it.

I also discuss the behaviour of specifications (also those spanning a number of institutions) under institution representation in some "universal logic". It turns out that adequate translation of specifications along an institution representation can be done syntactically provided the representation satisfies some simple model expansion and amalgamation properties. This allows one to move entire work with specifications over a diagram of institutions to the "universal institution" in which all of them are represented — provided that the representations enjoy the model expansion and amalgamation properties and are mutually compatible with each other.

A Toolkit for Requirements and Design Engineering

Roel Wieringa

Universiteit Twente

This talks sketches the outcome of research into integrating structured and object-oriented software design notations with each other and with formal specification. The result is a conceptual toolkit called TRADE (Toolkit for Requirements and Design Engineering), which is used for teaching software design to informatics students in a non-partisan way, that is without entrenching the students in a structured or object-oriented ideology. TRADE assumes a systems engineering framework, that views any system as part of a systems hierarchy that can be extended upwards or decomposed downwards. At each level, systems are specified by describing their external functions, behavior and communication. Structured and object-oriented methods offer techniques to specify these aspects: Decomposition is specified using class diagrams, functions are specified by means of event-response lists, behavior by means of state machines and communication by means of sequence diagrams. It can be shown that data flow diagrams are superfluous once state machines with local variables are used. Structured and object-oriented methods also offer decomposition heuristics such as function decomposition, domain-oriented decomposition and others. It can be shown that these can be combined. Some of these techniques and heuristics are essentially informal, others can be formalized. We have formalized class diagrams as well as ways to specify functions and behavior, using order-sorted dynamic logic with equality. These ideas are illustrated by means of a case study in which a controller for a compact dynamic bus station is specified.

Algebraic Models of CCS-Specifications

Uwe Wolter

Technische Universität Berlin

Usually CCS is understood as a calculus to specify and reason about processes. Thereby the concepts machine, state, and process are considered as synonyms. We take the viewpoint that all three concepts are different: A machine has different states and a process has two parameters — the machine where it takes place and the state where it starts.

The main observation outlined in the talk is that a specification written in the full value-passing calculus of CCS can be directly seen as a partial algebraic specification of machines/algebras. In particular it turns out that CCS is focussed on initial semantics of those specifications.

Based on this observation we present an algebraic semantics of CCS specifications in terms of combined machines/algebras. Then we point out that from the algebraic viewpoint process expressions provide a sophisticated language to speak about computations in machines/algebras and thus about behavioural equivalence of states in one machine and of different machines, and we relate this to traditional behavioural algebraic specifications.