UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Incremental Formalization: a Key to Industrial Success

Bernhard Steffen, Tiziana Margaria, Andreas Claßen Volker Braun

Lehrstuhl für Programmiersysteme
Universität Passau
Innstr. 33
D-94032 Passau (Germany)
tel: +49 851 509.3090
fax: +49 851 509.3092
{steffen,tiziana,classen,deepthou}@fmi.uni-passau.de

MIP-9516
September 1995

Contents

List of Figures

1 Motivation

In his invited talk at TAPSOFT'95, Joseph Goguen claimed that practical success of formal methods is bound to a *tool-supported domain specific* paradigm (cf. [GoLu95]), which he characterized as follows:

1. *a narrow, well defined, well understood problem domain is addressed; there may already be a successful library for this domain,*
2. *there is a community of users who understand the domain, have good communication among themselves, and have potential financial resources,*
3. *the tool has a graphical user interface that is intuitive to the user community, embodying their own language and conventions,*
4. *the tool takes a* large grain *approach; rather than synthesizing procedures out of statements, it synthesizes systems out of modules; it may use a library of components and synthesize code for putting them together,*
5. *inside the tool is a powerful engine that encapsulates formal methods concepts and/or algorithms; it may be a theorem prover or a code generator; users do not have to know how it works, or even that it is there.*

The META-Frame environment [StMC95] developed in Passau arose exactly from the need for such tool support.[1] It supports programming with whole subroutines and modules as elementary building blocks on the basis of a specifically instantiated repository. Originally, the need arose from the problem of the construction of application-specific heterogeneous analysis and verifications tools on the basis of a repository of elementary tools and algorithms [MaCS95, StMC95b]. In the meantime it turned out that META-Frame has a much wider application profile, as will be illustrated by means of a completely different application: the construction of intelligent network (IN) services. This application, which is described in detail in Section 3, led to a very successful industrial cooperation with the Siemens Nixdorf Informationssysteme AG (SNI). In fact, a corresponding product, prototypes of which are currently presented by SNI on various Telecommunication Fairs, will be on the market in 1996. Its development required the collaboration of five differents sites (two in Munich, Berlin, Vienna and Passau), with Passau spending more than 6 man years in '95 on this project, involving 17 students, researchers and post-docs.

Incremental formalization, ranging from 'no specification', which results in the old-fashioned development style, to 'detailed specification' with full tool support, was the key to the acceptance of our underlying method in an industrial environment.[2] However there were also other important factors for acceptance, like e.g. the strong support of software reuse (Sec. 4), enhanced flexibility (Sec. 5), an intuitive specification language (Sec. 6), graphical interface and hypertext support (Sec. 8), which we will discuss along the

[1]Other systems fitting this description are: CAPS [Luqi93], ControlH and MetaH [Vest94], AMPHION [SWLP94], Panel [ScSn94], and DSDL [BeKi94, KBBH95].

[2]This observation is in full correspondence with Christiane Floyd's statements at TAPSOFT'95 [Floy95].

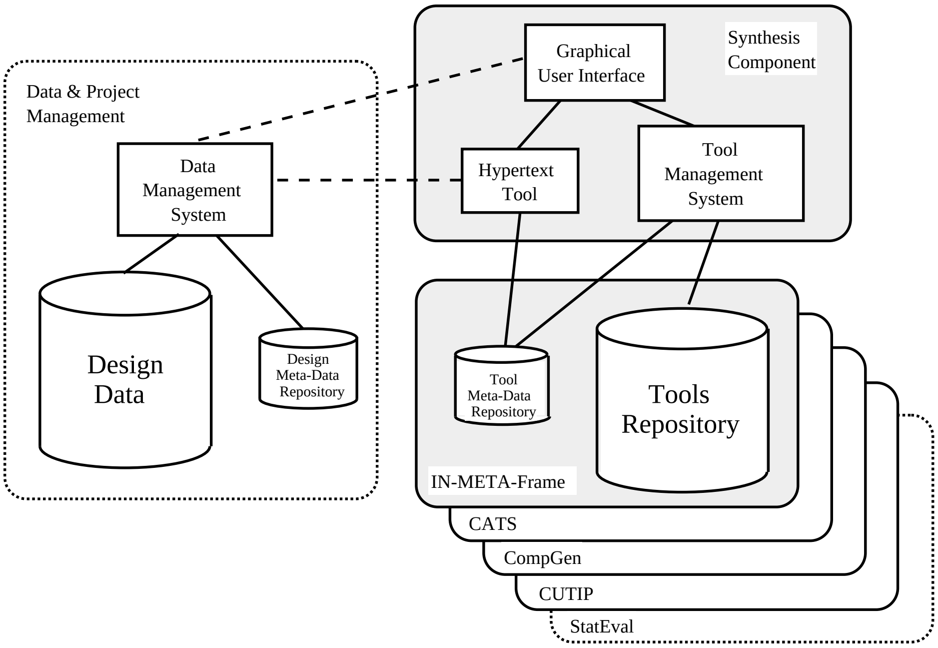

Figure 1: The META-Frame Global Architecture

lines of the "Ten Commandments of Formal Methods" proposed by Bowen and Hinchey in [BoHi95].

After sketching the basics of META-Frame, and of its industrial application in the next two sections, we present our philosophy and design decisions in Sections 4 to 9. As this discussion complements the brief sketch of our scenario, a full picture of our industrial application will be provided at the end. Section 10 summarizes our experience and perspectives.

2 The META-Frame Environment

META-Frame is a *meta-level environment* for the flexible management of large repositories of complex objects which are best modelled and described at a coarse-grain level. It is designed to offer sophisticated support for the *systematic* and *structured* computer aided generation, analysis, verification, and testing of application-specific complex systems from collections of reusable components. Figure 1 shows its overall organization. Special care has been taken in the design of an adequate, almost natural-language specification language, of a user-friendly graphical interface, of a hypertext based navigation tool, in the automation of the synthesis process, and of the repository management.

This application-independent core is complemented by application-specific libraries of components, which constitute the objects of the design. The principle of separating the component implementation from its description is systematically enforced: for each application we have a distinct Meta-Data repository containing an abstract description of the components. The components themselves and their documentation are available in a different repository. This organization offers a maximum of flexibility since the core of the system is independent of the direct physical availability of the components (except for the execution, which is a different matter).[3]

The resulting META-Frame allows a designer team the rapid and reliable realization of efficient application specific complex programs without a sophisticated user interaction, and is therefore an ideal means for a systematic investigation and construction of adequate large-to huge grain systems which solve specific problems.

The META-Frame approach abstracts from implementation details by allowing designers a *high-level-development* of the programs:

- **Specifications** express constraints in a *temporal logic*, SLTL, that uniformly and elegantly captures an abstract view of the repository (see Section 6).
- **Implementations** are in a *high-level language* (HLL) tailored to express the combination of reusable library components stored in the repository, which are considered as atomic on this level. These components may well be written in different programming languages.
- **Constructed programs** are hierarchical, based on subcomponents, which may be written in different application languages (C, ML, C++) of different programming paradigms (imperative, functional, object-oriented). META-Frame also supports the automatic generation of components from modal logic specifications [StMF93], a feature currently not exploited in the IN application.

META-Frame is currently available for SUN SparcStations under UNIX, for PCs under LINUX, and for Siemens RM Machines under SINIX. Its graphical interface as well as the hypertext browser are built on top of the Tcl/Tk graphics library [Oust94]. Target language is the HLL, whose interpreter is implemented in C++.

3 IN-META-Frame: Definition of Intelligent Network Services

Intelligent Network Services are customized telephone services, like e.g., 1) ‘Free-Phone’, where the receiver of the call can be billed if some conditions are met, 2) ‘Universal Private Telephone’, enabling groups of customers to define their own private net within the public net, or 3) ‘Partner Lines’, where a number of menus leads to the satisfaction of all desires. The realization of these services is quite complex and error prone.

Our joint project with the Siemens Nixdorf Informationssysteme AG aims at the development of a new environment for the definition of such services based on META-Frame.

[3]Additional features like e.g. the advanced data and project management features (already offered e.g. by object oriented CAD frameworks as NELSIS [WoBD90]), could be integrated with little effort.

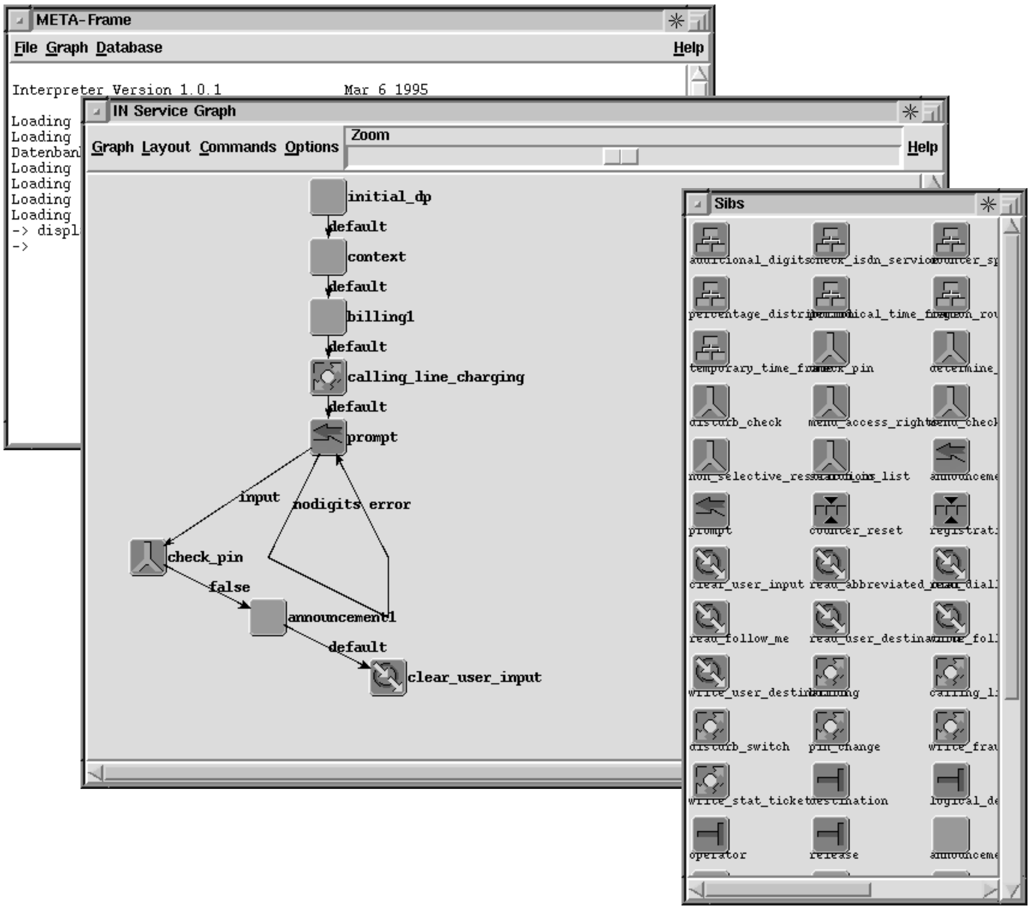

Figure 2: The *Service Logic Editor* with SIB Palette

Already in its current state, this environment is unique in offering global correctness and consistency checks guaranteeing frame conditions for the design concerning implementability, country specific standards and network specific features. Frame conditions are formulated as constraints in a temporal logic and verified via model checking. This approach provides an *incremental* use of formal methods: if no formal constraints are defined, our system behaves like standard systems for service definition. However, the more constraints are added, the more reliable are the defined services. This provides a 'soft' entry into the world of formal methods.

3.1 The Service Logic Editor

Our environment is used for the reliable, aspect-driven definition of Telephone Services in a 'divide and conquer' fashion: initial prototypes are successively modified until they satisfy the current requirements. The entire service definition process is supported by thematic views that focus on particular aspects of the service under consideration, i.e. aspects irrelevant for the current design decision are hidden. Moreover, the service definition is constantly accompanied by on-line verification of the validity of the required properties and of the executability conditions for intermediate prototypes: design decisions that conflict with the general frame conditions or requirements of the intended service are immediately detected via model checking.

The design activity is entirely carried out within the META-Frame-based IN-Service Logic Editor (SLE), which is used for the definition, display and modification of IN models. SLE windows (see e.g. Fig. 2 middle) allow developers to build, modify, verify, and execute IN models and views. Commands, programs, and functions of META-Frame and its applications are available via pull-down menus.

Granny's Free-Phone, the toy service shown in the Figures 2, 4, and 6 in different stages of definition, is a simple kind of Free-Phone (800-service), designed for grannies who would like to incentivate friends and relatives to call them by paying their calls if they are done at a convenient time of the day.

3.2 Benefits

The current trend in advanced Intelligent Network telecommunication services clearly evolves towards decoupling Service Processing Systems from the switch network (see e.g. [CrKu94]). The reasons for this tendency lie in the growing need for decentralization of the service processing, in the demand for quick customization of the offered services, and in the requirement of rapid availability of the modified or reconfigured services. Service Processing Systems are those elements of the Intelligent Network architecture which provide *service processing logic and control* but not *connection control and management*, which are provided by the underlying switch system. In particular, they include Service Definition environments, which are interactive environments responsible for the definition, modification, customization and provision of new services.

To be successful, decentralization requires the simplification of the development process (currently, the construction of a single service may take several 'expert years') as well as the decentralization of the (major part of) the quality control, because otherwise the bottleneck of centralization would strike back again. Thus the issue of estimating the cost of the introduction of the new SLE[4] is of minor importance here, as the standard environments fail to satisfy the practical needs. Moreover, as will be discussed in more detail in Section 7, the incrementality of our approach allows a smooth entry to the new technology avoiding drastic changes, which would be costly and lead to acceptance problems.

Summarizing, the major benefit we expect is a drastic reduction of the development and testing time (and thus of the costs) of new IN services. Experience showed that time to market is the crucial criterium to survive and prosper in a highly competitive market: fast delivery of innovative services is often a k.o. criterium.

[4] *III) Thou shalt estimate costs*

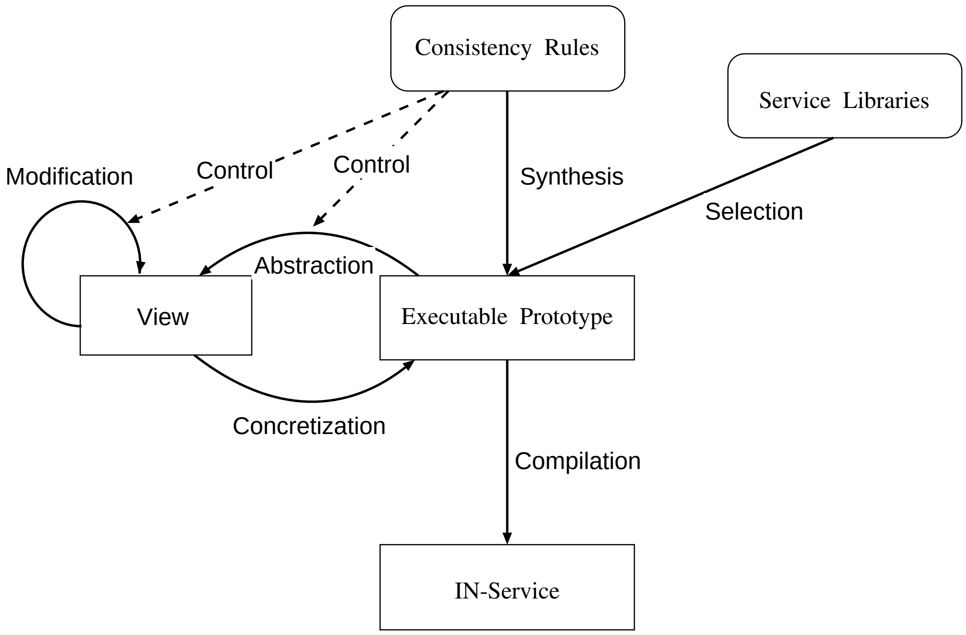

Figure 3: The Service Definition Process

4 Reusability

The key principle to development time and cost reduction is well known;

X) Thou shalt reuse

The "*four major factors which conspire against software reuse*" of [BoHi95] do not arise in the considered scenario. IN services are composed of procedure-like elementary components, the so-called Service Independent Building blocks (SIBs). Thus we are immediately confronted with a large-to-huge grain software engineering task. Of course, reuse is not only supported on the SIB-level, but also on the (sub-) services-level, leading to a hierarchical design. META-Frame is tailored for this style of development by flexible retrieval routines and hypertext support during the construction and the update of the underlying component repository [StMF93].

5 The Applied Formal Methods

Currently, our Service Definition environment integrates methods for logical specification, abstract interpretation, and formal verification in terms of model checking. A synthesis component is under construction. Figure 3 summarizes the architecture, which supports the arbitrary decomposition of the Service Definition process as long as the following three phases are preserved.

1. In a first step, an existing service of similar application profile is loaded from the service library, or a completely new design from scratch is done (under model checking

control). Alternatively, initial executable prototypes could automatically be generated from the constraints formulating the general frame conditions or requirements of the intended service, a feature, which is not part of the current version of our SD environment.

2. The second step consists of aspect-driven modification: the user chooses the aspect of interest, generates the corresponding view abstracting from all irrelevant details, and modifies it where necessary. This is iterated until all relevant aspects have been treated. Due to the on-line verification with the model checker, the executability is preserved and erroneous design steps are immediately detected.

3. Current prototypes can at any time be tested, compiled, executed, and, if satisfactory, stored in a data base.

As will be explained in the following sections, our environment is unique in guaranteeing the following three properties:

- during the entire definition process there exists a current *executable prototype* that can be tested and animated,
- the support through thematic *views*, which provide the required global context and hide unnecessary details, allows the designer to choose a particular aspect of interest, and to develop and investigate the services under that point of view. This supports a much more focused service development, which concentrates on the design of the aspect currently under investigation,
- the *global consistency* of each design step with implementation-related or service-dependent frame conditions is *automatically verified*. Thus sources for typical failures can be immediately detected.

It should be noted, however, that this covers the service definition only, which is only a small part of the whole development and implementation of a service. There it must also be dealt with the repositories or with the impressive software and hardware systems needed to run the service definition and management under real-life conditions.

The next sections are devoted to the single aspects of the environment.

6 The Metalevel Specification Language

The practical success of a specification language depends on an appropriate choice of notation[5], on the flexibility to allow loose specifications [6], and, additionally, on the availability of adequate tool support ([GoLu95]).

[5] *I) Thou shalt choose an appropriate notation*

[6] *II) Thou shalt formalize but not overformalize*

We use a natural language-like macro language based on a linear-time variant of Kozen's mu-calculus [Koze83], which comes together with efficient verification tools based on model checking:

- *properties* express correctness or consistency constraints the target IN service is required to respect. They are compiled into the temporal logic SLTL (Semantic Linear Time Logic, cf. [StMF93]), which is a sublogic of the mu-calculus, and
- *models* are the service logic graphs, where SIB names correspond to atomic propositions, and branching conditions correspond to action names in the SLTL formulas.

This specification language is tailored to express global properties of correct or implementable services structures, like, e.g., 'this SIB requires a certain other SIB to follow eventually' or 'a certain SIB excludes the use of another group of SIBs, unless something else holds'. This seems to be exactly the kind of constraints design experts inherently use. Our specification language allows us to make this inherent knowledge precise and available to an automatic checking procedure, which forces untrained designers to obey the essential 'rules of the game'. The library of such constraints is expected to grow soon to a size of up to 500 or even 1000 constraints. Using the model checker of [SCKK95], consistency of a service with the constraints can be checked fully automatically in real time.

Of course, the efficiency of the checking procedure also reveals that the specification language is less expressive than most of the specification languages usually used in software engineering: they are typically undecidable and the corresponding verification procedures require complex user interaction based on detailed knowledge of the involved algorithms. In contrast, our specification language allows fully automatic checking and even synthesis procedures ([StMF93]) shifting the user interaction to the *object level*, i.e. the level where the user's intuition lies [StMC95b]. In our experience this feature drastically increases the acceptance by practitioners.

7 Standard Features of the SLE

Service Definition environments for the design of IN services are usually based on classical 'Clipboard-Architecture' environments, where services are graphically constructed, compiled, and successively tested. Accordingly, 'must feature' are SIB palettes providing designers with a display of the available components, as shown in the window of Fig. 2 (right). Two extreme approaches characterize here the state of the art: The first approach guarantees consistency, but the definition process is strongly limited in its flexibility to compose SIBs to new services. The second approach allows flexible compositions of services, but gives little or no feedback on the correctness of the service under definition during its development: the validation is almost entirely located after the design is completed. Thus the resulting test phase is accordingly lengthy and costly.

Reliability control at the design level is usually limited to animation of the services (also called red-line tracing), which helps detecting design or configuration errors via *symbolic* *execution*. Unfortunately, exhaustive symbolic execution even of small size services like the Granny's Free-Phone is not feasible, because of the size of the underlying data domains. In spite of this, many large services (more than a hundred SIBs and several hundreds of edges) have had to be validated with this technique. This turned out to be very expensive and not to support a satisfactory notion of correctness.

According to the commandment

> *V) Thou shalt not abandon thy traditional development methods*

our environment supports these traditional design styles and allows a smooth transition towards the use of formal methods. In fact, it is possible to have developers in the same team, using the same tools, talking about the same objects,
but with a different 'formal background'. Thus the step towards a formal description (i.e. the introduction of new constraints) can be driven simply by need. E.g., whenever an inconsistency is found in a service, a corresponding constraint excluding such services should be added to the constraints library. Moreover, red-line tracing can be called at any time during the service definition, in order to improve or confirm the intuition about the behaviour of the service under development. This is and will be particularly important when dealing with customers. However, we expect that the role of red-line tracing as a validation technique will be replaced by formal verification in near future.

In the Granny's Free-Phone example, after a call initialization section common to all services, the caller dials the granny's specific freephone number, then he is prompted to enter a personal identification number (PIN) and, depending on the time of the day, the call is either released (in the forbidden time windows) after an announcement, or it is routed to the desired destination number. The complete service logic is shown in Fig. 4, together with a symbolic run. Clockwise,

(a) the user is asked to complete the simulation environment by input of the call's time and

(b) of the special number the Granny's Freephone corresponds to, then he is

(c) asked to give his Personal Identification Number, which is found to be incorrect.

(d) After the corresponding announcement, he gets

(e) a new chance (max 3 times). This time the PIN is correct, and since the time of the call lies within the allowed time intervals,

(f) the call is routed to Granny's actual phone number, and

(g) the simulation run terminates.

Figure 4: Symbolic Execution via Red Line Tracing

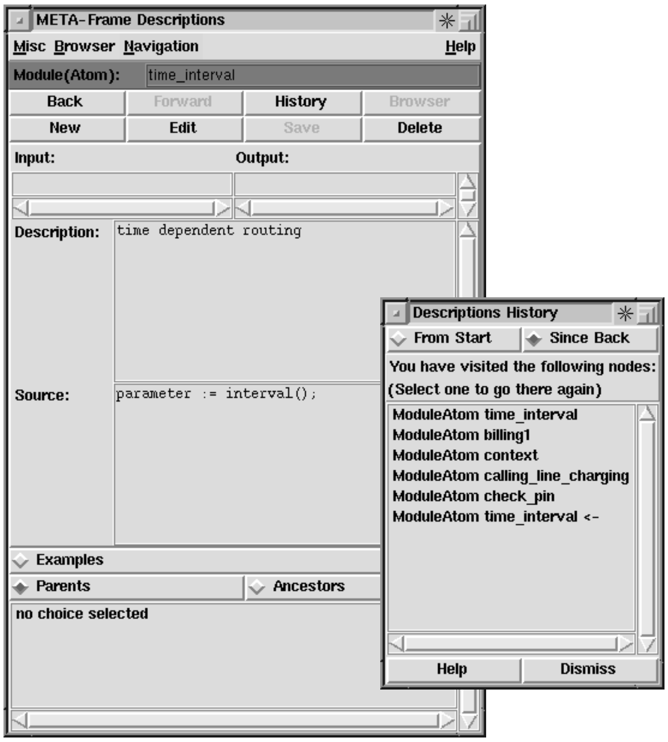

Figure 5: A SIB Hypertext Descriptor and the Browser's History

8 Hypertext: Help and User's Manual

Depending on the user's access permission, an on-line documentation via hypertext offers at any stage of the service definition comfortable information about central aspects of the system, including, e.g., the windows, menu entries (help), SIBs, services, and constraints. The hypertext system is activated from the SLE window and, in case of information about the SIBs, it presents a hypercard like the one of Fig. 5 (left). According to the principle

VI) Thou shalt document sufficiently

the many fields of an hypertext card for SIBs cover a variety of documentation aspects: a descriptive textual documentation on the module (field Description) is accompanied by its characteristics for retrieval, by some application examples (field Examples), and by the HLL code which is executed during animation via red-line-tracing (field Source). All these fields may contain hyperlinks which allow users to browse through the hypertext system using it as an online User's Manual. Navigation functions (step back, consult the history of visited cards as in Fig. 5 (right), step forward in the history), as well as editing functions (open a new card, edit or delete the current card) are easily accessed by means of the corresponding buttons.

Since hypertext cards are defined as collections of fields, they are configurable within the hypertext system by simple enabling/disabling of the fields. This allows a full customization of the information according to interests and permission of each user.

9 The Innovative Features

Our approach aims at structuring and guiding the design process in order to increase reliability: ensuring correctness can be much more than

IX) Thou shalt test, test, and test again.

In addition to the traditional animation, simulation, and testing, META-Frame offers the use of *abstract views* to support design decision and error correction, *formal verification* via model checking, and *synthesis from a constraint-based specification.* As the third aspect is not yet implemented in the IN application of META-Frame, we will concentrate only on the first two aspects in the following, and refer the interested reader to [StMC95b]. Of course, testing on the final product is still necessary in order to capture aspects orthogonal to the logical constraints. However, the correctness guaranteed by our formal verification allows a focussed testing of the remaining properties.

9.1 Abstract Views

Views are abstract service models, concentrating on certain aspects of actual, concrete models. They are used to hide any aspect of an IN model which is irrelevant for the intended operation, like fixing the way the service is billed or how the user-interaction proceeds. Views attack the problem of growing size of the services, which may contain several hundreds of nodes, and which are in their whole unmanageable. Particularly important applications are error views supporting error correction (Sec. 9.2), and the realization of the access control mechanism sketched below.

Operationally, views do not differ much from the actual IN models. E.g. they can be loaded and edited in the usual way, however, often with quite dramatic effects: minor modifications on views may correspond to radical structural changes of the underlying concrete model. In addition, views can be *edited* and then *applied* to the underlying concrete model, where application corresponds to the *concretization* function in Fig. 3. The interplay between abstract views and their concrete model will be illustrated in the next section. *Execution* of a view means execution of the underlying concrete model. It implicitly requires the application of the view.
Our concept of views generalizes the standard *stepwise refinement* approaches of the usual tools for Service Definition, since in particular also the decomposition into subservices has its associated view. In addition, views support the realization of a very flexible *access control mechanism*, by allowing designers and service providers the definition of customer specific views with restricted modification potential. Views provide in fact a natural division of the central design process (2nd step) into levels. In particular this allows us to tailor the environment for the specific needs of the service designer, the service provider,

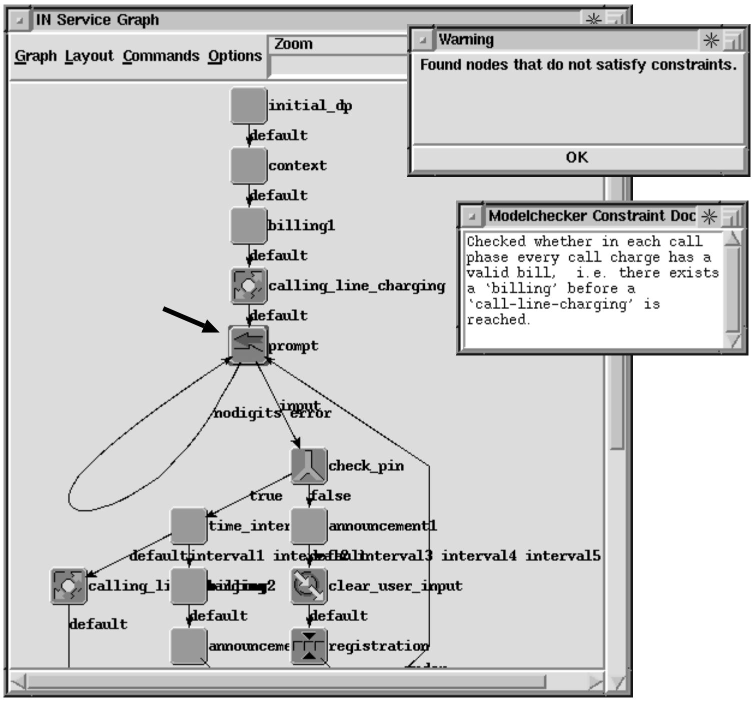

Figure 6: The Model Checker Finds a Billing Error

the customizer, and the user. The view-specific hiding can be used to automatically take care of access permissions.

9.2 Formal Verification

Orthogonal to the prototype animation, our environment provides automatic verification with respect to formal specifications. We distinguish two kinds of correctness checks:

- *local* correctness checks, guaranteeing e.g. the correct parametrization of single SIBs,
- *global* correctness and consistency of the configuration, concerning the interplay between the SIBs of a service.

While local checks are quite straightforward to implement, and are present also in other Service Definition environments, our environment is unique in offering global correctness and consistency checks, which can be verified at any time via model checking in a push-button fashion.

Global properties are expressed in a user friendly specification language (see Sec. 6), and gathered in a constraint library, which is automatically accessed by the model checker during the verification. The maintenance of the constraint library is supported by the hypertext system.

We use the model checker of [SCKK95], which is optimized for dealing with large numbers of constraints, in order to allow verification in real time. If the model checker detects

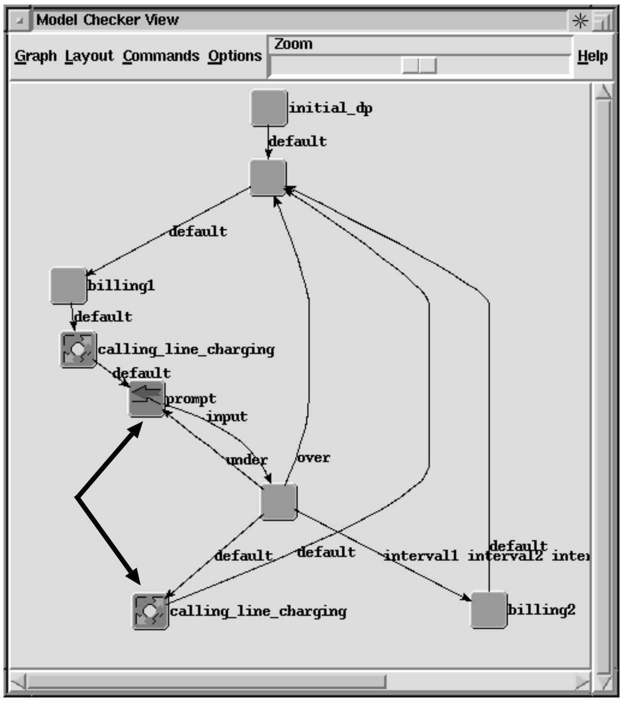

Figure 7: The Corresponding Error View

an inconsistency, a plain text explanation of the violated constraint appears in a window as shown in Figure 6, and an *error view* is automatically generated, concentrating on those nodes (SIBs) which are relevant to detect the error: all the other SIBs are collapsed to a single node (see Fig. 7), thus simplifying the location of the error. Corrections can be done directly on the error view, and the subsequent view application automatically updates the underlying concrete model.

In our example, model checking the service shown in Fig. 6 results in the discovery of erroneous paths in the graph, which violate the constraint shown in the same Figure in the rightmost window: On each segment (phase) of the calls, a bill should be issued before the system can charge for the segment. The problematic part of the paths starts at the prompt node marked by the arrow, whose icon is framed in red.

We are now going to correct the design error with the help of an error view.

- The error view of Fig. 7 has been obtained automatically after the failed model checking. Irrelevant nodes are collapsed in the unlabelled nodes. It is now easy to see that the path between the prompt and the calling-line-charging SIBs evidenced by the arrows contains a calling-line-charging which has no corresponding billing node. The location of the error is much more easily spotted.

- Fig. 8 shows the results of the error correction: the view graph has been edited, by inserting the billing node marked by the arrow in an appropriate position.

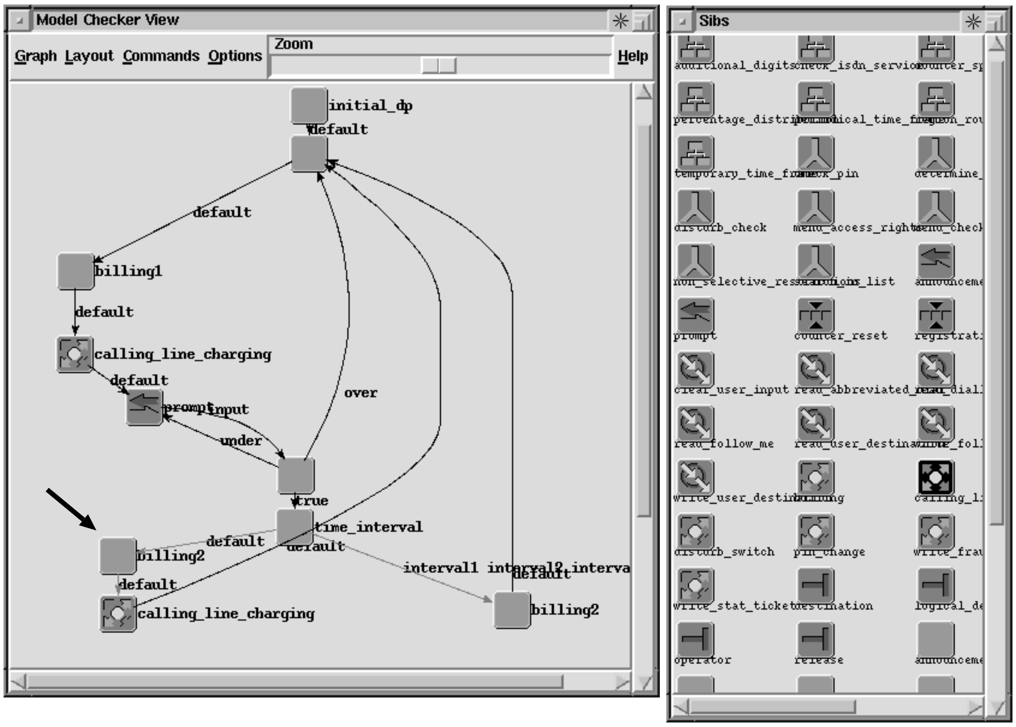

Figure 8: The Corrected Error View

- The Apply view command actualizes the corresponding IN-service model (see Fig. 9) which is thus automatically corrected. A new verification via model checking confirms the correctness of the corrected service.

Of course the simple example of the Granny's Free-Phone could have been easily handled by hand, but current IN services have reached sizes and complexities which demand for automated support for error detection, diagnosis, and correction.

10 Conclusions, Experience and Perspectives

Our approach exactly meets the demands recently expressed by Goguen and Luqi in [GoLu95] for the emerging paradigm of *Domain Specific Formal Methods*: The essence of their proposal is to use formal methods on a large or huge grain level rather than on elementary statements, thus to support the programming with whole subroutines and modules as elementary building blocks. This is precisely what META-Frame is designed for and what the application to Intelligent Network Services embodies.

Still, the step towards the use of formal methods is rather big and can hardly be done at once. Our environment therefore offers a *lazy* and *incremental* use of formal methods: if no formal constraints are defined, our system behaves like standard environments for

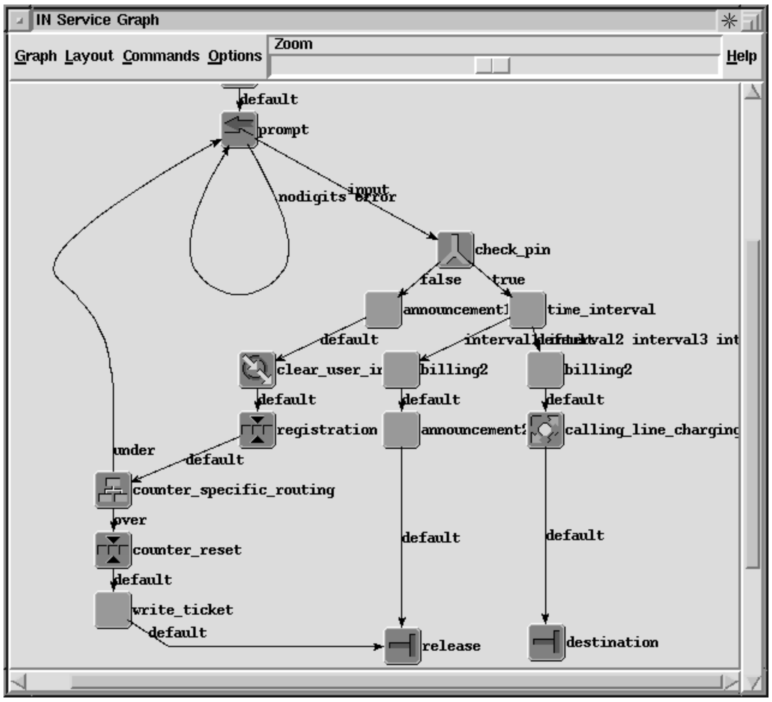

Figure 9: The Correct Granny's Freephone Service

Service Definition. However, the more constraints are added, the better is the support via automatic control and the more reliable are the defined services. In fact, in the extreme case of a full specification of the vital frame conditions, very few representative tests are required to guarantee the correctness of the service.

In our experience, it is often impossible to write complete formal specifications in industrial practice. E.g. in the case of Service Definition, the knowledge about the exact requirements is distributed over several groups along the development process, and it is only used implicitly during the development. Thus experts need to learn that and how this knowledge should be made explicit, e.g. via constraints. Our approach of incremental formalization takes this situation into account by providing a 'soft' entry into the world of formal methods.

The formulation of constraints will be in the hands of the university for some while[7], in particular when it comes to the limits of expressibility and to proposing alternative ways to complement the current constraint specifications by other methods in order to increase the overall expressive power. Currently, our group is responsible for the whole lifecycle of the formal methods portion, covering both the conceptual side and its implementation.

Besides the vast amount of integration into a 'real life' industrial product, the industrial partner also takes care of the industrial quality standard of the final product by

[7]Our experience agrees with the observation *IV) Thou shalt have a formal methods guru on call.*

guaranteeing certification according to the ISO 9000 standard.[8]

Finally, the flexible response to new demands[9] first convinced our industrial partner to go for META-Frame, and it is also the key to the project continuation. As concrete plans are confidential, we close by summarizing other applications of META-Frame also present in Figure 1 illustrating its flexibility:

- CATS: automatic generation of complex tools from a repository of components [MaCS95, StMC95b],
- CompGen: determination of the correct ordering of the individual analyses and optimizing transformations in a data flow analysis/program optimization setting [KnRS95],
- CUTIP: automatic composition of complex UNIX piping sequences for text and image processing [SFCM94, SFCM95] (DaCapo project, in cooperation with FORWISS[10]),
- StatEval: automatic generation of tools for the flexible evaluation of statistic data (in preparation).

Acknowledgements

We are grateful to Michael von der Beeck, Achim Dannecker, Philipp Florschütz, Carsten Friedrich, Andreas Holzmann, Marion Klein, Dirk Koschützki, Gerald Lüttgen, Falk Schreiber, Markus Schweighofer, and Matthias Seul for their cooperation in the design, implementation, and testing of META-Frame. We also thank the Siemens Nixdorf IN team and the GMRS team for their precious interaction in the definition of the characteristics of the system and for their valuable contributions to the realization of the current product.

[8] *VII) Thou shalt not compromise thy quality standards*

[9] *IX) Thou shalt not be dogmatic* was actually the key to the common project

[10] Bavarian Research Institute for Knowledge Based Systems

References

[BeKi94] J. Bell, R. Kieburtz, et al.: "*Software design for reliability and reuse: a proof-of-concept demonstation*", Techn. Rep., Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, Portland (OR), 1994.

[BoHi95] J. Bowen, M. Hinchey: "*Ten Commandments of Formal Methods*", IEEE Computer, April 1995, pp.56-63.

[CrKu94] E. Crabill, J. Kukla: "*Service Processing Systems for AT&T's Intelligent Network*", AT&T Technical Journal, Vol.73, N.6, Nov./Dec.1994, pp.39-47.

[Floy95] C. Floyd: "*Theory and Practice of Software Development - Stages in a Debate*", (invited talk) 6th Int. Conf. on Theory and Practice of Software Development (TAPSOFT'95), Aarhus (Denmark), May 1995, LNCS N.915, pp.25-41.

[GoLu95] J.A. Goguen, Luqi: "*Formal Methods and Social Context in Software Development*," (invited talk) 6th Int. Conf. on Theory and Practice of Software Development (TAPSOFT'95), Aarhus (Denmark), May 1995, LNCS N.915, pp.62-81.

[KBBH95] R. Kieburtz, F. Bellegarde, J. Bell, J. Hook, J. Lewis, D. Oliva, T. Sheard, L. Walton, T. Zhou: "*Calculating Software Generators from Solution Specifications*", 6th Int. Conf. on Theory and Practice of Software Development (TAPSOFT'95), Aarhus (Denmark), May 1995, LNCS N.915, pp.546-560.

[KnRS95] J. Knoop, O. Rüthing, B. Steffen: "*The power of assignment motion*," Proc. of the ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation (PLDI'95), La Jolla, California, June 18-21, 1995, SIGPLAN Notices 30, 6 (1995), pp.233-245.

[Koze83] D. Kozen: "*Results on the Propositional μ-Calculus*", Theoretical Computer Science, Vol. 27, 1983, pp. 333-354.

[Luqi93] Luqi: "*Real-time constraints in a rapid prototyping language*", Journal of Computer Languages, 18(2), 1993, pp.77-103.

[MaCS95] T. Margaria, A. Claßen, B. Steffen: "*Computer Aided Tool Synthesis in the* META-Frame ", 3. GI/ITG Workshop on "Anwendung formaler Methoden beim Entwurf von Hardwaresystemen", Passau (Germany), March 1995, pp. 11-20, Shaker Verlag.

[Oust94] John K. Ousterhout: "*Tcl and the Tk Toolkit*," Addison–Wesley, April 1994.

[SCKK95] B. Steffen, A. Claßen, M. Klein, J. Knoop. T. Margaria: "*The Fixpoint-Analysis Machine*", Proc. 6th Int. Conference on Concurrency (CONCUR'95), Philadelphia (PA), August 21-24, 1995, LNCS N. 962, pp.72-87.

[ScSn94] J. Schwartz, W. Kirk Snyder: "*Design of languages for multimedia applications development*", Proc. of the 1994 Monterey Workshop: Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development, pp. 46-55, 1994.

[SFCM94] B. Steffen, B. Freitag, A. Claßen, T. Margaria, U. Zukowski: "*Intelligent Software Synthesis in the* DaCapo *Environment*," Proc. 6th Nordic Worksh. on Programming Theory, Aarhus (DK), Oct. 1994, BRICS Report N.94/6.

[SFCM95] B. Steffen, B. Freitag, A. Claßen, T. Margaria, U. Zukowski: "*An Approach to Intelligent Software Library Management*", Proc. 4th Int. Conf. on Database Systems for Advanced Applications (DASFAA '95), National University of Singapore, April 10-13, 1995.

[StMC95] B. Steffen, A. Claßen, T. Margaria: "*The* META-Frame*: An Environment for Flexible Tool Management*," 6th Int. Conf. on Theory and Practice of Software Development (TAPSOFT'95), Aarhus (Denmark), May 1995, LNCS N.915, pp.791-792.

[StMC95b] B. Steffen, T. Margaria, A. Claßen: "*Heterogeneous Analysis and Verification for Distributed Systems*", Techn. Rep. MIP 95-09, Fakultät für Mathematik und Informatik, Universität Passau (Germany), July 1995.

[StMF93] B. Steffen, T. Margaria, B. Freitag: "*Module Configuration by Minimal Model Construction*," Techn. Rep. MIP 93-13, Fakultät für Mathematik und Informatik, Universität Passau (Germany), December 1993.

[SWLP94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood: "*Deductive Composition of Astronomical Software from Subroutine Libraries*", Proc. 12th Int. Conf on Automated Deduction (CADE'94), Nancy (F), June 994, LNAI N. 814, Springer V., pp. 341-355.

[Vest94] S. Vestal: "*Integrating control and software views in a CACE/CASE toolset*", Proc. Symposium on Computer-Aided Control Systems, 1994.

[WoBD90] P. van der Wolf, P. Bingley, P. Dewilde: "*On the Architecture of a CAD Framework: The NELSIS Approach*," Proc. EDAC'90, IEEE Computer Society Press, pp.29-33.