Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

T. Murer, A. Würtz, D. Scherer, D. Schweizer

GIPSY: Generating Integrated Process support Systems

Project Overview

TIK-Report
No. 22, December 1996

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

T. Murer, A. Würtz, D. Scherer, D. Schweizer
GIPSY: Generating Integrated Process support Systems - Project Overview
December 1996
Version 1
TIK-Report No. 22

Computer Engineering and Networks Laboratory,
Swiss Federal Institute of Technology (ETH) Zurich

Institut für Technische Informatik und Kommunikationsnetze,
Eidgenössische Technische Hochschule Zürich

Gloriastrasse 35, ETH-Zentrum, CH-8092 Zürich, Switzerland

ABSTRACT
Cooperative software engineering requires integration of a distributed and heterogeneous world of actors and resources. Component technology will augment integration problems due to distributed development of large numbers of interoperating components. Extensible Attribute Grammars (EAG) are used for the specification and generation of highly integrated tool components that support the software process.

Tool components are plugged into a process support framework which allows for the distributed definition and enactment of a software process. A common understanding of the process is provided by a 3D process model which supports tool integration on a high level so that developers perceive the tools as a homogeneous environment.

Keywords
Integrated Cooperative Software Engineering, Tool Components, Data Integration, Attribute Grammars, Software Process

INTRODUCTION
The goal of a software process is the development or maintenance of a software product. This process takes place in a complex, distributed and heterogeneous world of actors and resources (people, computers, tools, process data, network services, etc.), which have to be integrated and coordinated. An IPSE (Integrated Process Support Environment) [5] is a set of tool components that are plugged together to form a cohesive environment. Its strength lies in supporting the more formal tasks of cooperation within a process. This includes formal process definition, enactment and control, version and configuration management. While the process members' technical competence, creativity and their capability to cooperate in an informal way still remain very important to the success of a process [18], IPSEs support the execution of a software process efficiently and are essential for processes of a certain complexity. Nevertheless, IPSEs are not as widely accepted by software developers as they should be. One reason is the fact that the user's perception of an IPSE is not as homogeneous as it should be, because there are many aspects of tool integration.

Emerging Component Ware
Software component technology will change the way software is produced [1]. A set of heterogeneous interoperating components will replace a monolithic application. Different component producers need to cooperate in order to develop properly interoperating components. Producers weaken their enterprise boundaries by sharing and linking their development processes resulting in tightly cooperating networks of enterprises, so-called *virtual enterprises*. This, together with the globalization and increasing pace of software development requires highly integrated software development environments.

Tool Integration
'Integration' is a popular buzzword for the marketing of software systems. However, integration properties must be assessed carefully, since integration can have various meanings.

Different Kinds Of Integration
The interaction of tools in an IPSE is based on common integration services. A reference model for frameworks was established by ECMA [4] (European Computer Manufacturer's Association). This model (fig. 1) includes services such as messaging, control integration, data integration, repository and presentation integration, and influences existing IPSE implementations [17].

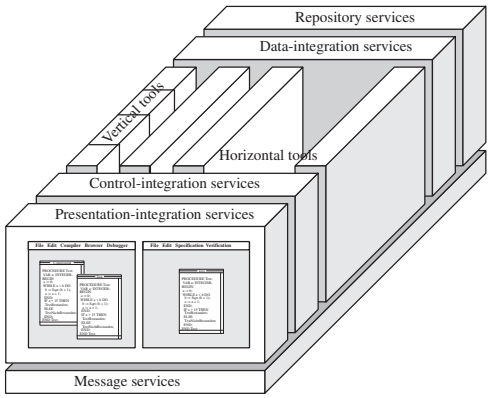

Fig. 1: Reference Model for IPSE Frameworks

Message and presentation integration can be built using today's standards for distributed objects [14] and GUI frameworks. Data integration and an intuitive, cohesive appearance of the IPSE require additional investigation.

Different Integration Levels
Besides integration kinds the degree of integration is also important, as the following example from the body-care domain shows. If you plug your American electric razor (power supply 110V/60Hz) into a European socket (220V/50Hz) using an appropriate adaptor, you reach integration on a mechanical level, but the razor is destroyed when switched on. An advanced model may have a voltage switch for a higher integration level. But if the blade swing frequency is a multiple of the AC frequency, the blade swings slower than in the US and shaving takes longer. Integration on the desired level requires a more sophisticated model. Similar problems occur in the computer domain when software development tools are integrated.

THE GIPSY PROJECT

The GIPSY (**G**enerating **I**ntegrated **P**rocess support **SY**stems) project [6] targets the domain of safety critical systems where the use of formal development methods is appropriate. Its goal is a fast transformation of results from computer science and software engineering methodology into modern tools. There are two main areas of focus:

- The CHIPS (**C**omponents for **H**ighly **I**ntegrated **P**rocess **S**upport) project deals with fine-grain steps of process support. Its goal is data integration on the highest possible level. Compiler-compiler technology is extended by the object-orientation paradigm to define and generate extensible tool components and their interoperation.
- The DIPS (**D**istributed **I**ntegrated **P**rocess **S**ervices) project investigates the coarse-grain steps of process support. It provides an IPSE framework that enables distributed process definition, enactment and tracing. The goal is an intuitive and cohesive appearance of the framework towards tools and users.

The simplified development path (fig. 2) shows how tool components are plugged into the framework in order to build an IPSE.

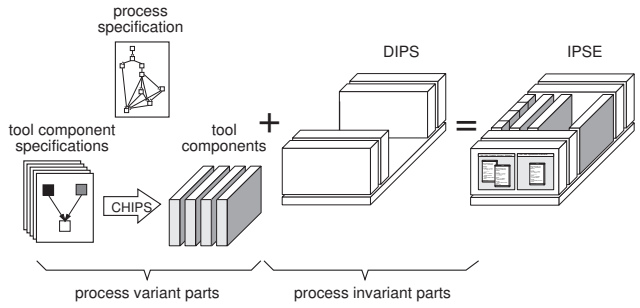

Fig. 2: Simplified Development Path of an IPSE

CHIPS

Data Integration Levels

Data integration problems occur when different tools provide different views and operations for the same type of data (fig. 3, tools A and B). Different tools typically implement their proprietary internal data structure. An arrow object, for example, which is drawn in a graphical editor and copied into a text editor is broken up into a line object and a triangle object. This incomplete integration leads to unsatisfactory printouts.

Tools working on two different data types can also cause integration problems (fig. 3, tool C). A graphical programming tool, for example, generates C source code from a state diagram and automatically invokes a C compiler for the generation of executable code. The user can develop a system on a graphical abstraction without knowing anything about C. If the graphical tool and the C compiler do not share exactly the same understanding of the C language, the C compiler may generate error messages which are useless for the user.

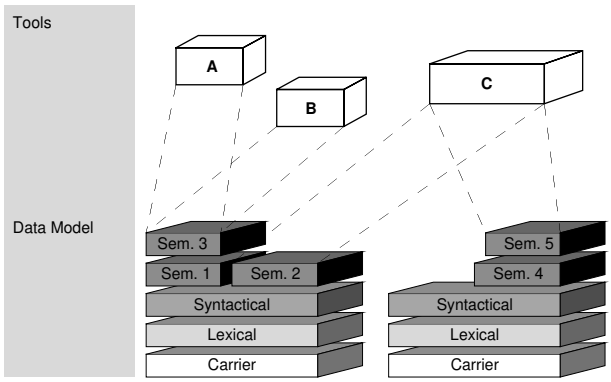

Fig. 3: Data Integration Levels

It is important to which degree tools share the same understanding of data semantics. In order to assess the degree of integration, a pragmatic classification of data integration into levels has been introduced [2]:

- Carrier level: Sharing the same set of primitive information units (e.g. ASCII character set)
- Lexical level: Sharing the same set of words (e.g. keywords in a programming language)
- Syntactic level: Sharing the same syntactic structure (e.g. sentences of a programming language)
- Semantic levels: There can be various semantic levels (e.g. declaration analysis, typechecking) defined on top of other levels

Tool integration is defined on data interchange formats that are often restricted to the structural aspects of data description. There are only limited possibilities to express data semantics and valid operations, restricting integration to the syntactic level.

How to achieve high data integration

To achieve a higher degree of data integration, tools must share data structures instead of implementing proprietary ones. This can be done by strictly separating tools into data models and views.

This results in the advantage of views being much more flexible because they only implement visualization techniques and are decoupled from data models. To the user this means the ability to dynamically configure an individual combination of view components from a toolkit.

The separation imposes special requirements on the data model. It has to be an abstract data type (ADT), which provides a complete description of the structure and semantics of the data, as well as the operations that are possible.

Goals of the CHIPS project
The goals of the CHIPS project are:

- Find a language that allows the specification of text-based data models having the properties stated above.
- Implement a generator that generates a tool component from this specification. Tool components are ADTs as mentioned above.

The CHIPS Data Model
Attribute Grammars
If data are restricted to textual formal languages, a well-suited ADT to represent them is a syntax tree. From an EBNF description (grammar) of a language's context-free syntax, a parser for the generation of a syntax tree can be easily generated using compiler-compiler techniques. Let `Calc` be a language consisting of integer additions, subtractions and equations, such as the following:

```
1+1      3+2=7
```

The grammar for such a language would look as follows:

```
GRAMMAR Calc;

NONTERMINAL
   Equation ::= Expression ['=' Expression].

NONTERMINAL
   Expression ::= integer {('+'|'-') integer}.

END Calc.
```

Many paradigms for the implementation of attribute grammars have been discussed [15]. For CHIPS a variant of object-oriented approach was chosen. It is implemented in such a way that every nonterminal description is treated as a class definition. The grammar in the example above introduces a class `Equation` and a class `Expression`. Every node in the actual syntax tree is an instance of such a class (fig. 4).

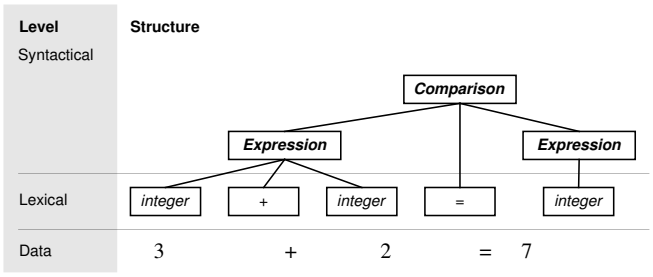

Fig. 4: The CHIPS Data Model

So far the ADT covers carrier, lexical and syntactic levels. Data is actually stored as plain text (carrier level), a lexical (token list) and a syntactic (syntax tree) structure is generated at load time.

Attributes
In an attribute grammar nonterminals can be annotated with attributes and attribute evaluation methods to implement semantic language properties. In CHIPS attributes are class fields and attribute evaluation is done by class methods. A certain semantic can be evaluated by walking over the syntax tree and calling the corresponding evaluation method of every node in the tree.

To facilitate attribute evaluation, an abstract node class *SynNode* has been defined, that provides methods to navigate within a nonterminal's production. Every nonterminal class is derived from *SynNode*.

FirstNT, NextNT, PrevNT, LastNT: deliver the first, next, previous and last nonterminal object inside the production of the calling nonterminal object.

FirstTerm, NextTerm, PrevTerm, LastTerm: deliver the first, next, previous and last terminal object belonging directly to the production of the calling nonterminal object. Fig. 5 shows how the terminal- and nonterminal navigation routines work.

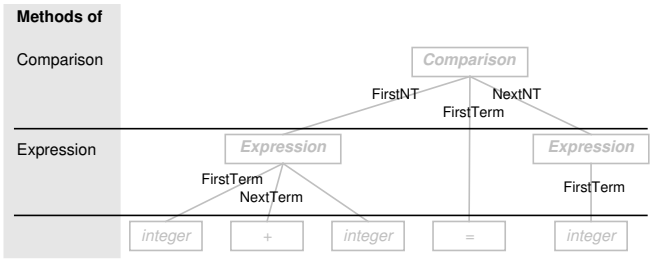

Fig. 5: Navigation on the Syntax Tree

The example below shows how the attribute `value` of the nonterminal `Expression` from our example grammar is evaluated using the navigation methods:

```
NONTERMINAL
   Expression::= integer {('+'|'-') integer}.

ATTR value: INTEGER;

PROCEDURE (self: Expression)Eval;
VAR
   number, op: TERMINAL;
   sum: INTEGER;
BEGIN
   number:=self.FirstTerm();
   sum:=StringToInt(number.val);
   op:=self.NextTerm(number);
   WHILE op#NIL DO
      number:=self.NextTerm(op);
      IF IsTerm(op, '+') THEN
         sum:=sum+StringToInt(number.val)
      ELSE
         sum:=sum-StringToInt(number.val)
      END;
      op:=self.NextTerm(number)
   END;
   self.value:=sum
END Eval;
```

So far the grammar also allows mathematical equations that are not correct. A method `Validate` might test if the nonterminal `Equation` represents a valid equation using the expression's `Eval` method as shown below. This method adds a mathematical semantic. `3+2=7` is now not a valid sentence any more.

```
NONTERMINAL
   Equation::= Expression [`=` Expression].

PROCEDURE (self: Equation)Validate;
VAR
   expr1, expr2: Expression;
BEGIN
   expr1:=self.FirstNT();
   expr2:=self.NextNT(expr1)
   IF expr2#NIL THEN
      expr1.Eval;
      expr2.Eval;
      IF expr1.value#expr2.value THEN
         ERROR("Equation is not true")
      END
   END
END Validate;
```

Extensible Attribute Grammars

Why Extensibility?

Extensibility of data models is required for the following reasons:

- *Extending Semantics:* A new tool may not only provide new visualizations, but also new operations to an existing tool component. With a strict separation of a tool into view and tool component, the new operations have to be implemented through an extension of the existing tool component. Existing views can still visualize extended components. If two tools are extensions of the same component, they share a common understanding on the level defined by their common base component. Thus both can perform operations and exchange data on that level.
- *Decomposition:* Complete descriptions of tool components tend to be large. In order to make them more readable, the monolithic descriptions need to be decomposed. Extensibility allows adding semantics to a component without invalidating existing semantics. Thus monolithic descriptions may be split into semantic layers. Decomposition has for example been used successfully during the specification of CHIPS. At the moment CHIPS is able to generate itself, which was implemented in three steps: language syntax, semantic analysis and generation.
- *Interoperability:* Of vital importance for highly integrated tool components is the capability to describe how they cooperate. Relations between two components can be established by implementing an extension of one component that uses the other component. Access to the used component is restricted to the operations it offers. Thus a high level of integration is guaranteed.
- *Abstraction:* A good example of reuse by extension is the abstract definition of the *SynNode* class described above. It only defines the behavior, not the syntactical structure of a SynNode, which is added later in a grammar definition. Reuse possibilities can be widely extended by defining abstract grammars. These grammars only contain abstract nonterminals, i.e. they have a behavior but no syntactical definition. For example, all block-oriented languages provide a hierarchical scope mechanism which defines where objects can be declared and accessed from. It would be very convenient to have an abstract grammar *BlockGram* defining a nonterminal *Scope* that is not yet syntactically specified by a production, but defines a scope semantic that allows the declaration and referencing of abstract objects. Any grammar for a concrete block-oriented language could then be derived from *BlockGram* and every nonterminal that opens a scope would be an extension of *BlockGram.Scope.* A browser allows inspecting the implementations of declared objects. If it was implemented for *BlockGram,* it would instantly work for all block-oriented languages, leading to reusage in the implementations of both the data model and the view.

How to achieve extension?

There are several possibilities for the extension of a grammar [10]. While the extension of a language's syntax is thinkable, it would be quite difficult to handle. To meet the requirements above, the following extension mechanism is proposed. An extension preserves the syntactic structure of the language and existing semantics, but it defines additional semantics and operations. A new tool component is defined by the extension of a grammar. The extension of a grammar consists of the extension of some of its nonterminals.

An example

As a simple example for the extension, a grammar `AddCalc` has been implemented below as an extension of the original Calc. `AddCalc` provides a new attribute evaluation `Inc` to the nonterminal `Expression` that increments the attribute `value` by the value of the expression every time it is called.

```
GRAMMAR AddCalc(Calc);

NONTERMINAL Expression(Calc.Expression)

PROCEDURE (self: Expression)Inc;
BEGIN
   previous:=self.value;
   self.Eval;
   INC(self.value, previous)
END Inc;
```

States

Attribute evaluations may be classified into *property evaluations* and *queries*. The successful evaluation of a property transforms the tool component into the corresponding *state* (mostly meaning a higher semantic level) while a query only provides information about the component. In order to be performed, every attribute evaluation requires the component to be in a certain state. There are also predefined states, such as *none* or *syntactically correct*.

The state of a tool component is kept as a list of successfully executed attribute evaluations. If the component is changed, all entries in its list are removed, returning it to the *none* state.

Interoperability

As mentioned above, a well-defined description of interoperability is of vital importance for highly integrated tool components. Relations between two components can be established by *importing* one component into the other. Access to the imported component is restricted to the operations it offers, guaranteeing the highest possible level of integration. The example below illustrates how `MaxCalc` (an extension of `Calc`) imports another Grammar `MaxGram` that defines a maximum value for each expression. The method `TestMax` checks if the expression value exceeds this maximum.

```
GRAMMAR MaxCalc(Calc)

IMPORT MaxGram;

NONTERMINAL Expression(Calc.Expression).

PROCEDURE (self: Expression)TestMax;
VAR
   max: INTEGER;
   M: MaxGram
BEGIN
   (* pseudo code *)
   M:=FindRelatedComponentOfType(MaxGram);
   SendMessageToComponent(M, 'GetMax', max);
   (* end of pseudo code *)
   self.Eval;
   IF self.value>max THEN
      ERROR("Expression value too high")
   END
END TestMax;

END MaxCalc.
```

Context management

If an object of grammar A needs to interact with an object of grammar B, it requires a mechanism to find such a corresponding object. So not only the relations between grammars have to be described but also the relations between actual instances of these grammars. In the example above, this operation was hidden by pseudocode. A way to give a tool component access to its contextual environment is described in the following section.

DIPS

A Common Understanding of the Software Process

Subtasks of the process are covered by different tools within an IPSE, each providing its own method and notation. A process control tool, for example, offers a petri net notation to define and visualize process structure and state. A configuration management tool uses version trees to manage data. A repository tool uses graphs to define data types and their relations. These tools may be integrated from the technical point of view, but they do not share a common understanding of the software process.

A common understanding of the software process means a common knowledge of its structure and evolution. All process participants (users and tools) should share this understanding and therefore it should be the underlying concept for tool design. It determines the effectiveness of the IPSE's process support and its attraction to potential users. The DIPS (Distributed Integrated Process Services) framework is designed to provide an intuitive and cohesive appearance of the whole IPSE towards users. Tool components (CHIPS) are plugged into the DIPS framework to build a cohesive IPSE that enables distributed integrated process support.

The following sections outline how processes are defined, enacted and traced and how components are plugged into the framework.

Working Hypothesis

All data (contract, specification, implementation, documentation, code, test case, manual, etc.) which are produced during development and maintenance of a software product can be represented as a partially ordered set of objects. Every object carries the information of a part of the software product and typically depends on other objects. These dependencies define the ordering of the set.

Software Process Notation

To use a directed acyclic graph (DAG) to define a software process is a rather intuitive approach, because the structure of the process matches exactly the expected structure of the planned software product. While designing the development of a software product, an *atomic process* (fig. 6) is assigned to every planned object, and the planned dependencies between the objects define the connections between the processes. The arrow at the end of each connection points in the direction of workflow, which is opposite to dependencies. During process enactment the desired objects have to be created and assigned to every process. For every process a predicate is defined that the assigned object must satisfy. Predicate definition and validation include preceding neighbor processes.

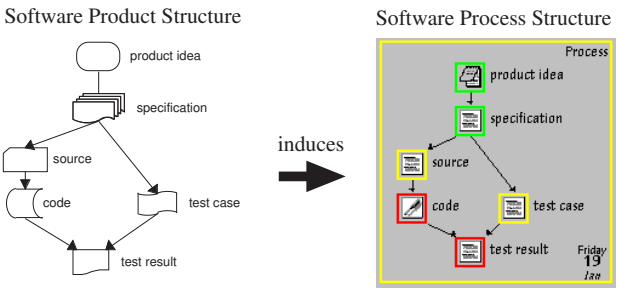

Fig. 6: Product Structure Defines Process Structure

Static Process Semantics

A process passes the sequence of the following states (indicated by different colours) exactly once:

- *planned:* no data object available yet
- *in work:* data object is available but does not satisfy the required predicate
- *confirmed:* data object is available and satisfies the required predicate

Hierarchic structures can be achieved by *compound processes* that can contain other processes. The state of a compound process is...

- *planned:* iff all contained processes are in state planned
- *in work:* iff it is not in state planned or confirmed
- *confirmed:* iff all contained processes are in state confirmed

The hierarchy introduces an abstraction with a defined semantic. On the next higher level of hierarchy a compound process behaves like an atomic process.

Alias Process Linking

Due to the emerging software component market the development of software that depends tightly on other software occurs frequently. It is of vital importance to connect different development processes on a high level. This can be done by *alias processes* which can be linked to any atomic process which is in state confirmed and part of another process hierarchy.

Dynamic Process Semantics

The process structure and the 3 states a process can be in define a *finite state space*. In the beginning all processes are in the state planned. The accepting state is reached if all processes are in state confirmed. State transition is defined by rules like: A process can change from state in work to state confirmed if all preceding processes are in state confirmed and the associated predicate is evaluated successfully.

The process state space may be changed during process evolution by adding or removing objects and dependencies.

Process Phases

Every process is in one of three phases. The sequence of these phases is defined by the process state sequence (fig. 7). The process may alternate between phases that belong to the same state.

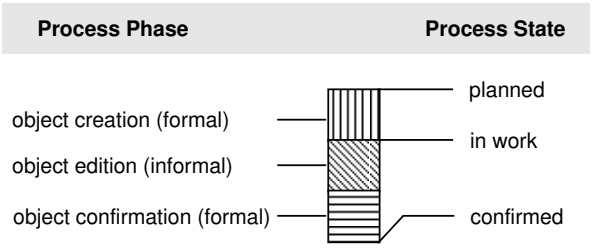

Fig. 7: Process Phases

Object creation: A data object is created and assigned to the process. A successful creation will transform the process state from planned to in work. The creation phase is defined by the *creation message* attribute of the process and is executed by the framework. The message is typically sent to an object placed on a preceding process. The receiver of the message will create and return an object (e.g. a source code object creates an executable code object) which will be placed on the process.

Object edition: The assigned object is edited in the creative edition phase. The framework will provide a suitable view to work on the object.

Object confirmation: A successful confirmation phase will transform the process state from in work to confirmed. The confirmation phase is defined by the *confirmation message* attribute of the process and is executed by the framework. The message is sent to the assigned object.

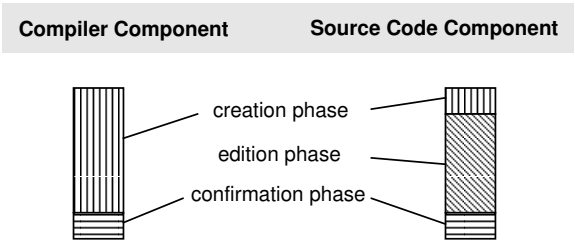

Fig. 8: Examples of Process Phases

Objects as Instances of Tool Components

Objects assigned to processes are typically instances of the tool components introduced above. Creation and confirmation messages are provided by tool components.

The importance of the different phases depends on the component type of the assigned object. The important phase with respect to a source code component is the edition phase wherein the source code is written. The creation phase may provide a template and the confirmation phase checks the correct syntax. The main phase of a compiler component is

the creation phase where executable code is created. An edition phase does not exist for obvious reasons (fig. 8).

Meta Process Linking
Tool components have their own development process. If an instance of a tool component is assigned to an atomic process, that tool component's development process becomes a meta process with respect to the atomic process. The use of the correct version of the supporting tool component is assured by a meta link which is established on the assignment of the creation and confirmation message.

A 3D Software Process Model
A process definition shows - in two dimensions - process attributes, process dependencies, possible concurrent workflow, process state and progress.

Third Dimension: Process History, Object Versions
An unpleasant but frequent step in the software process is the invalidation and revision of objects that were already confirmed. To keep track of these revisions the model provides a history dimension. Every time a process must be revised, a copy of it (without objects) with state planned is created and its older versions move down the history dimension. Since all depending processes have to be revised as well, a new layer is added to the process history, containing the revised processes and all its dependants (fig. 9). Different versions of the same process can be found by moving along the history dimension. This 3D model contains complete information about the structure, state and history of the process. It provides an overview containing useful state, dependency and version information for every object [13].

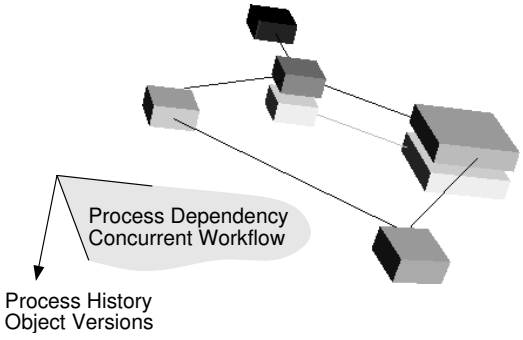

Fig. 9: Process Dimensions

This model could be the underlying concept for tools that are integrated on an understanding level. The screen dump of a sample process navigator (fig. 10) gives an idea how the model can be applied to adopt upcoming 3D visualization and virtual reality [11] into the software engineering domain.

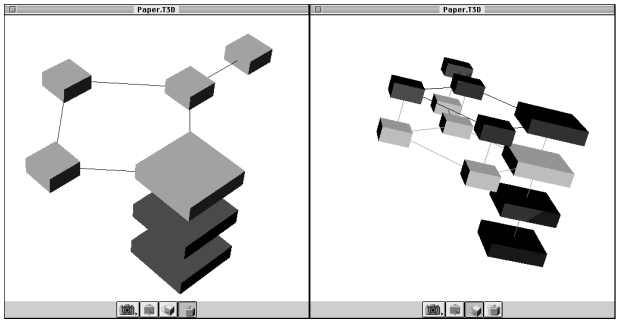

Fig. 10: Example of a 3D Process Navigator

Architecture
Distributed Objects
The software process takes place in a complex, distributed and heterogeneous world of actors and resources. This fact is reflected in the underlying object request broker (ORB) architecture of the DIPS framework. An object remains on the host where it was created for its whole life and uses messages forwarded over the network by ORBs to communicate with other objects. A CORBA-compliant [14] implementation of the ORB is envisaged.

Replicated Processes
Navigating through the 3-dimensional process space to obtain an overview of state, progress and history of a process is a frequent operation which should be supported efficiently by the underlying architecture. Therefore the whole 3-dimensional data model of a process is replicated on every host that participates in the process. Atomicity of operations and consistency of the replicated models are guaranteed by transactions and one dedicated master process.

Integration in the WWW
As an outlook, the process framework could also be integrated in the World-Wide Web (WWW), e.g. by utilizing a Java-based [8] CORBA-compliant ORB as the underlying architecture. This would allow participation in process operations via the WWW, independent of a particular implementation of the framework, facilitating widely distributed cooperative software engineering by providing an attractive homogeneous perception of heterogeneous process data while requiring little administration effort for participants on heterogeneous platforms.

As a more immediate application, the extraction of interoperability information of software components from their development processes and presentation of this information on the WWW has been investigated as a promising application [12, 16], which could be incorporated in a component catalog for world-wide electronic software component marketing.

VALIDATION

Bootstrapping the GIPSY Project

A prototype of the GIPSY environment has been implemented in Oberon System 3 [9]. The prototype is used in its own development process in a bootstrapping way. We have reached a milestone by defining and generating the prototype by itself. Figs. 11 and 12 show screen dumps of the CHIPS and DIPS parts of the prototype.

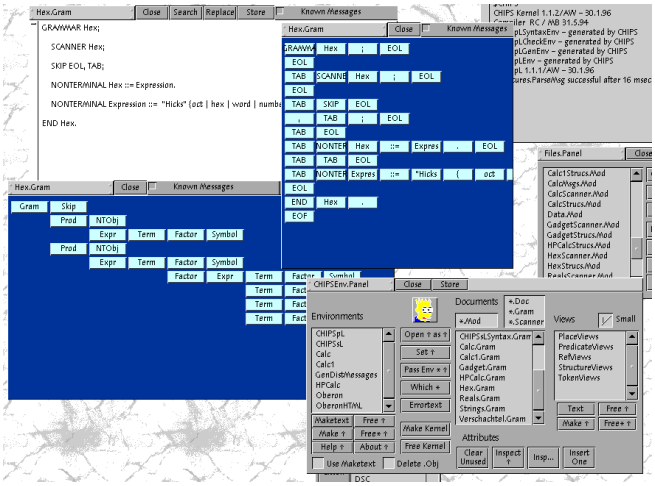

Fig. 11: Screen Dump of the CHIPS Prototype

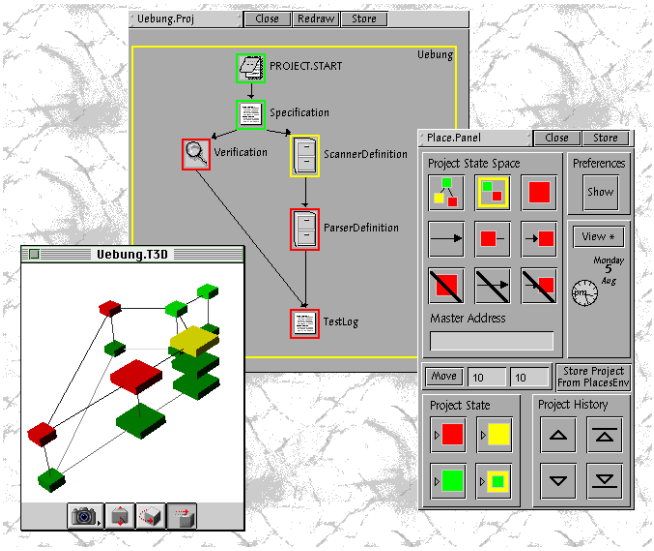

Fig. 12: Screen Dump of the DIPS Prototype

The PESCA Project

Formal verification of algebraically specified imperative programs is the main issue of the PESCA Project [3]. In earlier approaches, implementations of abstract data types by means of automatic code generation from algebraic specifications lead to poor results with respect to memory and time consumption. Due to that fact, efficient implementations still require a careful design, done by skilled software engineers. Following the design and implementation tasks, a verification session establishes the correctness of the implementation, i. e. shows that the assertions in the *Transformed implementation* imply the assertions of the corresponding algebraic specification.

We suggest the following software development process (fig. 13):

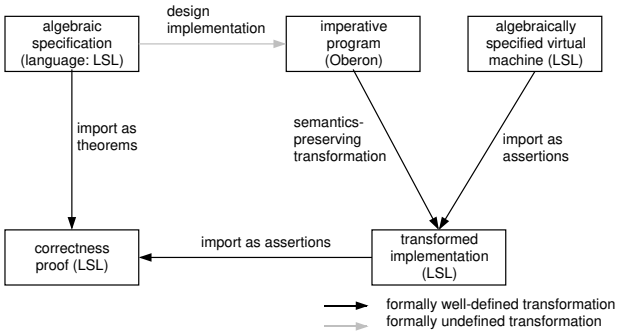

Fig. 13: Suggested Software Development Process

Every stage of the process is supported by appropriate tools. In the *Algebraic specification* and *Correctness proof* part, e.g., we use the LARCH [7] toolset. The *semantics preserving transformation* is accomplished by means of a CHIPS generated tool component. A high level of data integration between algebraic and imperative parts, along with well-defined context management, are of crucial importance to the successful application of our approach.

GIPSY in Education

The prototype is used in exercises of lectures at ETH. It also supports planning and progress of student projects.

CONTRIBUTIONS AND CONCLUSIONS

CHIPS

Tool integration is of vital and emerging importance for the efficient and effective support of cooperative software engineering. We illustrated how data integration can be achieved on the highest level. We combined compiler-compiler technology with the object-oriented paradigm into a prototype which enables the definition and generation of highly integrated extensible tool components. Interoperability of tool components can also be described in a proper way because data access is restricted to its provided operations.

Two dilemmas have to be kept in mind for the design of highly integrated tools:

- Data integration on a high level demands the strict separation between data model and view in the tool architecture. The stricter this separation is (i.e. the more abstract the data are), the more difficult it becomes to design new tools which add new semantics or operations to work on an existing data format, if no complete and extensible data description mechanism is present.

- Data integration on a high level demands the complete definition of structure, semantics and operations of a data model. Data model definitions should be easily readable and understandable by tool designers and tool users. The more complete a data model definition is, the less readable and understandable it becomes.

The extension mechanism of the introduced Extensible Attributed Grammars is a contribution to resolve the two dilemmas. New tools based on an existing data model are implemented by extending the data model. Complete data model definitions are not in a monolithic block, but split into several grammars which can be exchanged individually. Thereby grammars can be reused, are maintainable and easier to read and understand.

DIPS

Integration aspects are often restricted to the technical layer of tool implementation. The lack of integration between tools and their users is reflected in the tools and discourages developers from using them. We introduced a simple 3D model for a common understanding of the structure (2 dimensions) and evolution (3rd dimension) of a software process. It provides access to essential process properties like object content, dependency and version information in an easily perceptible and comprehensive way. The model should be the underlying concept of every tool combined in an IPSE that intends to attract users through a cohesive appearance.

The model provides abstraction from type, location, platform and tools of individual objects. Thus it is open to be implemented on top of an architecture of heterogeneous distributed objects.

The model could bring 3D visualization and virtual reality to a domain that usually does not consider these techniques (despite having much information). Since the model is not constrained to software development, this opens opportunities to other business processes or workflow applications. An interesting implementation of the model could be based on World-Wide Web technologies.

ACKNOWLEDGEMENTS

The GIPSY project has been funded in part by the Swiss Priority Programme (SPP) Informatics Research of the Swiss National Science Foundation.

REFERENCES

[1] Adler, R. M. Emerging Standards for Component Software. *IEEE Computer* (March 1995), 68-77.

[2] Brown, A. W., McDermit, J. A. On integration and reuse in a software development environment. *Software Engineering Environments* (Aberystwyth, March 1991), Horwood, 171-194.

[3] Denzler, C., Schweizer, D. PESCA: Programming Environment for Safety Critical Applications. <http://www.tik.ee.ethz.ch/~pesca/>, 1996.

[4] Earl, A. A Reference Model for Frameworks of Computer Assisted Software Engineering Environments. *Technical Report ECMA/TR /90/55*, ECMA, Geneva, December 1990.

[5] Finkelstein, A., Kramer, J., Nuseibeh, B. Software Process Modelling and Technology. RSP Ltd, 1994.

[6] GIPSY home page (with subprojects CHIPS and DIPS). <http://www.tik.ee.ethz.ch/~gipsy/>, 1996.

[7] Guttag, J. V., Horning, J. J. (ed). Larch: Languages and Tools for Formal Specification. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[8] Java home page. <http://java.sun.com/>.

[9] Marais, J. Oberon System 3. <http://huxley.inf.ethz.ch/~marais/Spirit.html>, 1995.

[10] Marti, R., Murer, T. Extensible Attribute Grammars. *TIK Report No. 6* (December 1992), Computer Engineering and Networks Laboratory, ETH Zürich.

[11] Megantz, B. QuickDraw 3D-The Future IS (Virtually) Here. *Apple Directions* (October 1995), 14-21.

[12] Murer, T., Scherer, D., Würtz, A. Improving Component Interoperability Information. *Proc. WCOP-96 (Workshop on Component-Oriented Programming, at ECOOP '96 (10th European Conference on Object-Oriented Programming))*, Linz, Austria, July 1996.

[13] Murer, T., Würtz, A., Scherer, D. A 3D Model for a Common Understanding of the Software Process. *Proc. APCHI 96 (Asia-Pacific Conference on Computer Human Interaction)*, Singapore, June 1996, 318-323.

[14] Orfali, R., Harkey, D., Edwards, J. The Essential Distributed Objects Survival Guide. Wiley, 1996.

[15] Paakki, J., Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys* (June 1995), ACM Press, 196-255.

[16] Scherer, D., Murer, T., Würtz, A. Towards Providing Software Component Interoperability Information on the WWW. *Proc. AusWeb 96 (2nd Australian World Wide Web Conference)*, Gold Coast, <http://www.scu.edu.au/ausweb96/tech/scherer/>, July 1996, 109-116.

[17] Sharon, D., Bell, R. Tools that bind: Creating Integrated Environments. *IEEE Software* (March 1995), 76-85.

[18] Whittaker, S., Schwarz, H. Back to the Future:.... *Proc. CHI'95* (Denver, May 1995), ACM Press, 495-502.