UNIVERSITÄT PASSAU

Fakultät für Mathematik und Informatik

Automatic Synthesis of Design Plans in METAFrame

Tiziana Margaria, Bernhard Steffen

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

MIP-9610
April 1996

Abstract

We present CAD-MetaFrame, an environment supporting the flexible CAD tool integration and application-specific construction of design plans, which avoids the insurgence of unsuccessful design plans at design time. During a planning phase the collection of all complete, executable design plans is automatically synthesized on the basis of simple constraint-like specifications and the library of available tools. The designer's choice of the best alternative is eased by a user friendly graphical interface and by hypertext support for the generation and management of plans, as illustrated along a user session.

Contents

List of Figures

1 Motivation

To master the complexity and variety of system-level design, flexible CAD environments are required, which efficiently support reuse, 'design in the large' strategies and teamwork. In particular, a rich collection of design tools with different application profiles is needed to allow an application-specific treatment of complex design tasks. CAD Frameworks, like e.g. NELSIS [WoBD90], CADLAB [MGSW89], or the Jessi CAD Framework [SNI93], have succeeded to provide a sophisticated and user friendly project management wrt. data and design flow, but still offer little support for advanced application-specific synthesis of design plans involving a flexible use of available tools. First attempts to combine heterogeneous verification systems with frameworks, like in PREVAIL-DM [Wagn93], limit themselves to a primitive tool management[1]. Some implicit tool management is also offered by VISTA [HPRS95], an integration and development system which enables the cooperation of different simulation tools for physical design. Generic tool management was first realized in design environments like ULYSSES [BuDi87], ADAM [KnPa86] and more recently Cadweld [DaDi91]. In these systems, tools are either directly chosen according to hardwired criteria or they are dynamically proposed at design-time according to the remaining design task. Recently, Hercules [BrDi95, JoBr95] and NELSIS [WoBH94] enhanced these methods to a powerful workflow management, which captures both tool and design management aspects. However, as a consequence of their treatment of design planning *during* design execution, all these systems detect unsuccessful design attempts *after* some unsuccessful design step has already been carried out, which requires complex backtrackings to undo wrong design decisions.

With CAD-MetaFrame we propose the introduction of a computer-aided planning phase prior to design execution, where design plans are generated from loose formal description of the desired design tasks. Core of our method is the *automatic synthesis* of the collection of all (all minimal or all minimal length) complete, executable design plans on the basis of simple but powerful constraint-like specifications and the library of available CAD tools. The distinctive feature of our specifications is their *global* character, i.e., in our context, their ability to express *interdependencies* between tools. Exactly such interdependencies are responsible for the expensive backtrackings at design time, since they cannot be captured by formalisms describing properties of single tools, and thus cannot be taken into account while proposing tools "on the fly" as in the common local approaches.

Being implemented on top of MetaFrame, a general purpose environment for the systematic and structured computer aided generation of application-specific complex objects from collections of reusable components, CAD-MetaFrame additionally supports the flexible tool integration into a CAD environment. Moreover its hypertext system, graphical user interface and local checking routines provide a strong basis for *local* design decisions. Thus the formalism behind the global planning can be introduced incrementally with CAD-MetaFrame: if no formal constraints are used, local design decision are supported,

[1]PREVAIL offers three tools (the TACHE tautology checker, the Boyer-Moore theorem prover for 1st order logic and the BDD-based LOVERT register-transfer level verification tool) which address different problems and therefore do not leave alternatives to the user.

however, the more constraints are used the better is the automatic support. In fact, playing with the synthesis algorithm by successively strengthening and loosing specifications, its feedback via the solution graph is a good guidance even during the requirement engineering phase (cf. Sec. 5).

The next section will give an overview of METAFrame, Section 3 presents the synthesis component, Section 4 defines our specification language SLTL, and Sections 5 and 6 illustrate the computer-aided synthesis of design plans.

2 The METAFrame Environment

METAFrame is a *meta-level framework* designed to offer a sophisticated support for the systematic and structured computer aided generation of application-specific complex objects from collections of reusable components. Special care has been taken in the design of an intuitive specification language, of a user-friendly graphical interface, of a hypertext based navigation and documentation tool, and in the automation of the synthesis process (our planner).

This application-independent core is complemented by application-specific libraries of components, which constitute the objects of the synthesis. The principle of separating the component implementation from its description is systematically enforced: for each application we have a distinct meta-data repository containing a logical view of the components, comprising information similar to the one in [DaDi91], but also links to a rich documentation. This information is accessible via hypertext. Components' implementations and their documentation are available in different repositories. This organization offers a maximum of flexibility since the synthesis core is independent of the direct physical availability of the tools and can work on a library including *virtual* components.

METAFrame [StMC95, SMCB96b] was already successfully used for the construction of heterogenous analysis and verification algorithms [StMC96], and in a joint project with Siemens Nixdorf Munich [SMCB95, SMCB96] for the design and implementation of IN-METAFrame, an environment for the creation of advanced telephone services (Intelligent Network Services). The corresponding product was presented by Siemens Nixdorf at the Telecom'95 in Geneva and at the CeBit'96 in Hannover, and has recently reached the market.

METAFrame is currently available for UNIX systems, e.g. the Siemens Nixdorf RM line, and for PCs under LINUX. Its graphical interface as well as the hypertext browser are built on top of the Tcl/Tk graphics library [Oust94].

3 Synthesizing Plans in CAD-METAFrame

The novelty of our approach consists of introducing a declarative specification layer, which is used for the construction of the desired plans according to *global properties* guaranteeing

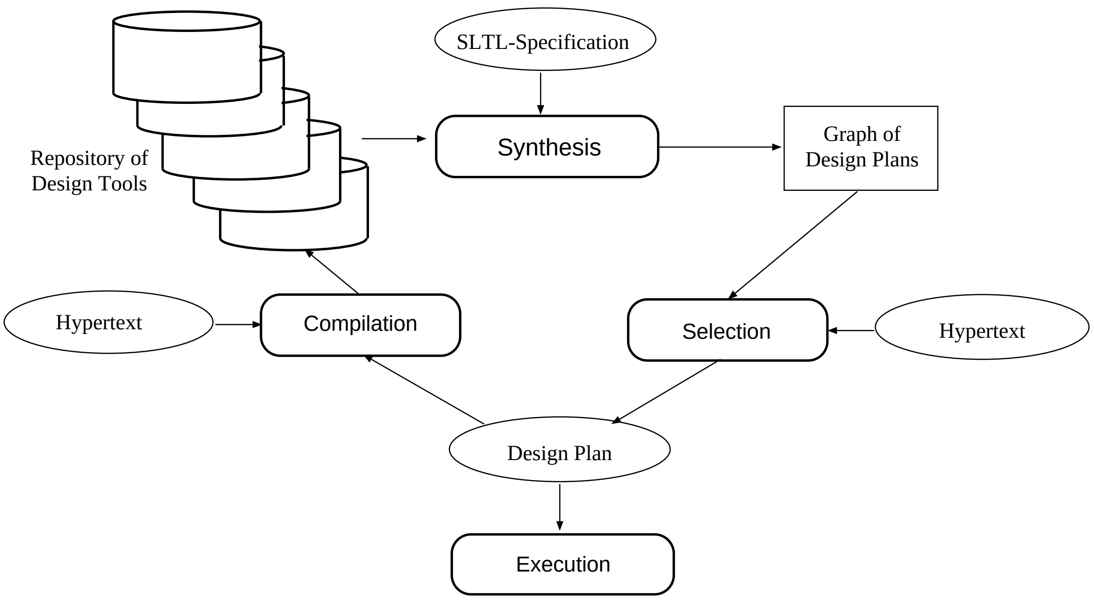

Figure 1: The Synthesis Process

executability, tool compatibility, and other consistency conditions.

CAD-MetaFrame can be regarded as a global planner where

- the **Repository** contains a collection of basic reusable components which are considered as atomic on this level. Components include elementary CAD tools, but also transformation components to bridge the gap between different representations or I/O formats.
- **Design Tasks** are (combinations of) instantiated reusable components available in the repository, where instantiation means that multifunctional tools are separately configured for each specific function.
- **Specifications** of design plans express constraints of admissible plans. They may be input in natural language-like form, which is automatically translated into a *temporal logic*, SLTL (see Section 4).
- **Synthesized plans** are hierarchically structured: design plans are themselves considered as (complex) design tasks on a higher level of the design hierarchy. Synthesized plans are directly executable, as they are written in a high-level programming language for the combination of complex component programs, which may even be written in different programming languages.

Figure 1 illustrates the interplay of the CAD-MetaFrame components during the synthesis process. Given a design-plan specification, our system synthesizes a graph representing the set of all (all minimal or all minimal length) tool compositions satisfying the specification, which can be built on the basis of the underlying repository. Each path is a design plan consistent with the specification. Proposed tool compositions may differ in the techniques they use, their application profiles, and, because of the potential looseness of specifications, in their overall behaviour. This overview allows designers to choose their most appropriate design on the basis of a complete graphical description of all possibilities offered by the underlying library of tools. Whereas global aspects of this collection are represented in form of a graph, we rely on hypertext support for 1) the investigation of the characteristics of single tools and data structures, 2) information on the current state of the tool repository, and 3) the classification scheme (called taxonomy) characterizing the application profile of each available tool.

Besides proposing the set of all successful plans, our system also provides diagnostic information highlighting the reason of failure of other plans. We will illustrate the features of our tool in Section 5 along the lines of the motivating example for Cadweld [DaDi91]. This allows a direct comparison with the most similar approach. A more detailed discussion of related work can be found in [BrDi95, StMC96].

Our synthesis component works by model construction roughly along the lines of [MaWo84]. In general *global constraints* lead to an exponential worst-case complexity, which is not surprising for a global search problem. However, typical specifications consist of a number of well-defined subgoals which must be met successively. These subgoals can be treated independently. Together with our limitation to linear design plans, i.e., plans which only require sequential compositions of tools, this leads to a good performance in practice. General compositions require user interaction for treating branches and joins, as described in Section 6 for the synthesis-based design plan editor.

Advanced data and project management features offered by object oriented CAD frameworks as NELSIS or Cadweld, or the task management of Hercules are orthogonal and could be integrated too.

4 The Metalevel Specification Language

Plans are generated by the synthesis component of MetaFrame by determining the set of solutions to specifications given in terms of global and local constraints. These constraints can uniformly and elegantly be formulated within the temporal logic SLTL (Semantic Linear-Time Temporal Logic), which combines *type*, *module*, and *ordering* constraints to a specification language with a coherent semantics (see [StMF93] for details). Given an SLTL specification, all legal (design task) compositions satisfying these constraints are synthesized.

CAD tools and algorithms are seen as transformations that take an input and produce the corresponding output. Thus they can abstractly be described as a triple (*input-type*, `operation`, *output-type*). There can be more than one input/output combination for an operation, thus allowing polymorphism[2].

[2]Each combination can be seen as a separate design task, i.e. an instantiation of the functionality of the tool.

Our goal is the specification and automatic synthesis of *legal plans*, i.e., of executable finite sequences of design tasks realizable within the underlying library. Obviously, two tasks can be sequentially combined iff they are *neighbours*, i.e. the output type of the first is compatible with the input type of the second. Legal plans are simply sequential compositions of successively neighbouring design tasks.

A core issue is the abstract characterization of the tools and algorithms available in the library. Each of them has an associated abstract description in terms of a taxonomic classification which establishes its application profile in terms of properties. Figure 3 shows the module taxonomy used in the following sections as presented by the hypertext system. It classifies the subset of tools available in the library that is relevant for the example. A *taxonomy* is a directed acyclic graph. Sinks represent concrete modules (labelled with the concrete tool names), which are atomic entities in the taxonomy, and intermediate nodes represent groups, i.e., sets of tools satisfying some basic property (expressed by predicates). Similarly, we have a taxonomy on the input/output types induced by the tools. This guarantees the flexible and efficient selectivity of the specifications and thus the good algorithmic performance of the synthesis component. Figure 2 shows the type taxonomy used in the example.

Our languages for expressing *local* constraints, capturing interfacing and single tool specifications, and *global* constraints, restricting the ordering between tools along a design plan, are presented in the next two paragraphs.

Local Constraints: The Static Aspect. Local constraints are the selection criteria for *single components*, i.e., single design steps allowed to appear in the final plan. They are based on the taxonomic classification of the repository components, whose design is the crucial part of the MetaFrame instantiation for a specific application. We distinguish

- *Interface* or *Type constraints*, which describe the 'neighbourhood' of single elements of the repository in terms of type compatibility. They range over the type taxonomy, and
- *Module constraints*, which restrict the set of legal elementary tools which may appear in the design plan. They range over the module taxonomy.

Both forms of local constraints are formulated as simple propositional logic formulas over the respective taxonomies, which are regarded as definitions of sets of basic predicates (atoms). As an example, in the module taxonomy of Figure 3 the module constraint relative to placement tools

`global and not robust`

individuates the set { `TimberWolfMC, TimberWolfSC_4, VPNR` }, which are tools for global placement that are not considered robust.

Similarly, in the type taxonomy of Figure 2 the type constraint

`cad_formats and hw_descr_lang`

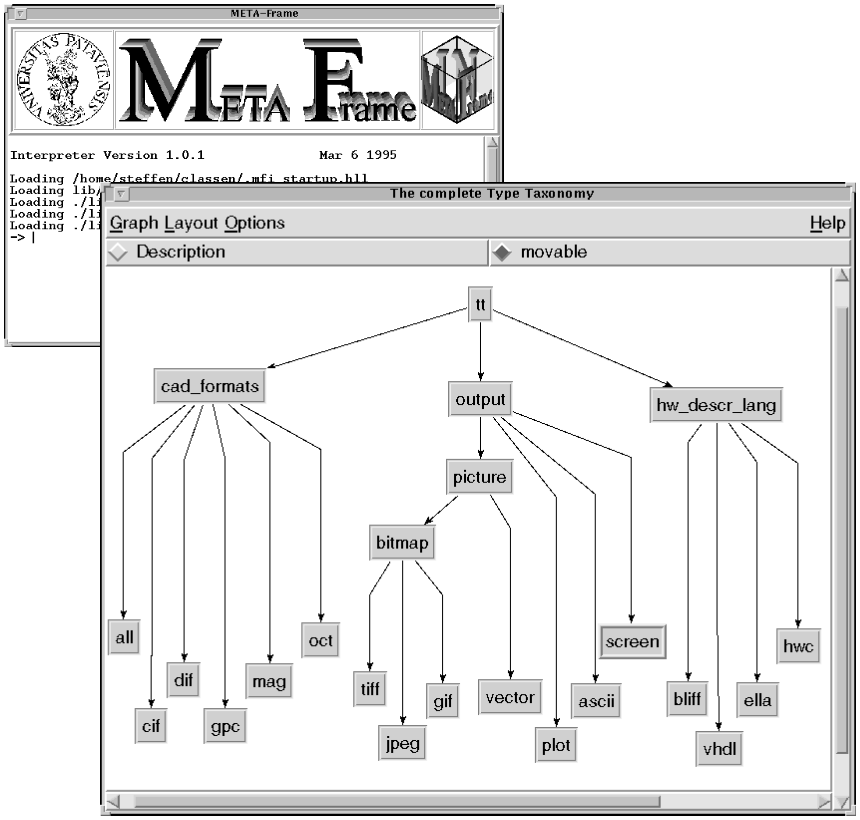

Figure 2: Main Window and Type Taxonomy

returns the empty set, since none of the types classified under `cad_formats` is also a hardware description language.

The *task schemata* approach in Hercules'workflow management [BrDi95] captures exactly local constraints. The information on functional and data dependencies between tools, there represented in a task schema, is contained in CAD-MetaFrame's module and type taxonomy. Hercules' task schemata are stepwise constructed by designers by means of successive *Expansion* and selection steps. (Type correct) expansion is equivalent to a type constraint for an atomic type, which retrieves all the neighbouring modules to a given one[3]. Selection corresponds to choosing one of them, which is then included in the task schema. Plans are therefore interactively constructed as type-correct successions of *single* modules.

The following paragraph describes the *innovative* features of our specification language, which allow the automatic construction of plans according to global constraints.

Global Constraints: The Temporal Aspect. Global constraints are the innovative feature of our planner. They allow users to specify a relative ordering between (some of) the components that form a solution. Using a temporal logic as a framework, we are able

[3]An example for a user's local constraint query in CAD-meta is contained in Section 5.

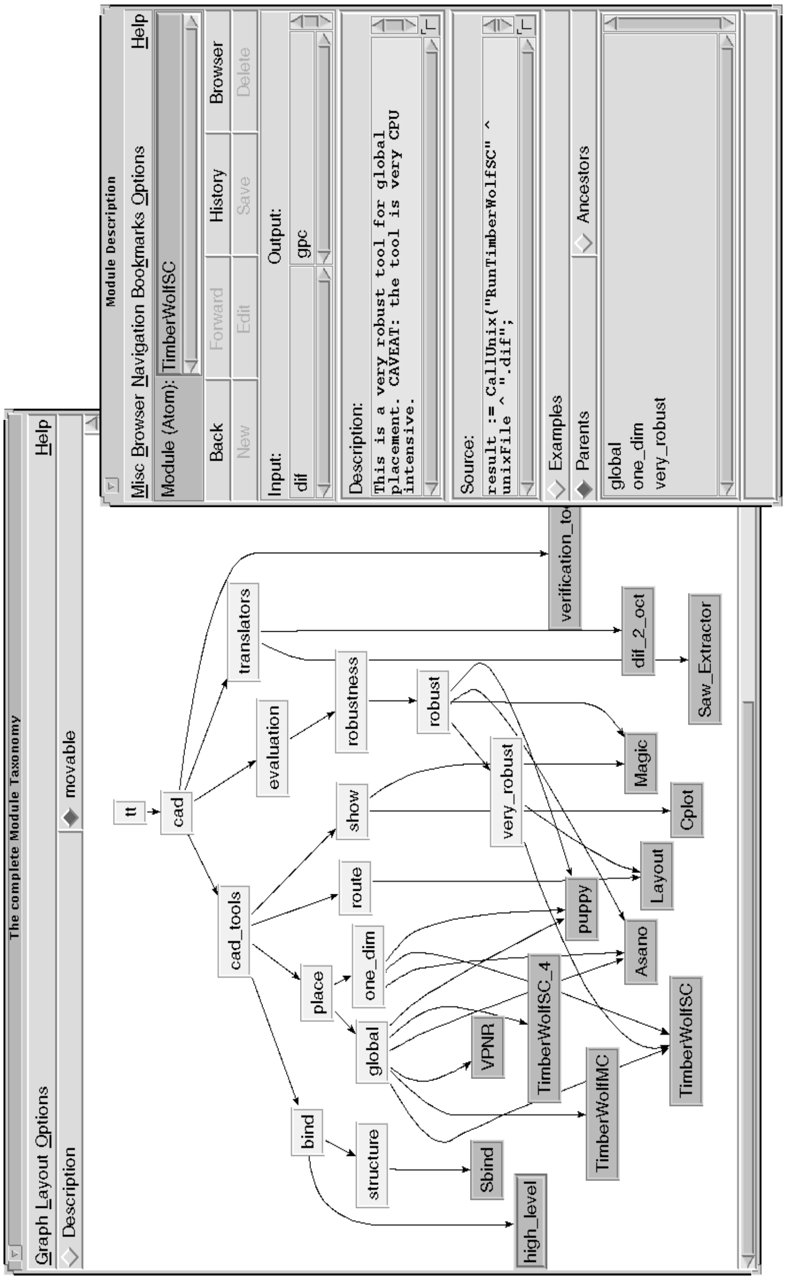

Figure 3: Module Taxonomy and a Hypertext Card

to extend the two local kinds of specifications to an elegant and uniform specification language for global constraints, which is defined as follows:

Definition 4.1 (SLTL) *The syntax of Semantic Linear-Time Temporal Logic (SLTL) is given by*[4]*:*

$$\Phi ::= \quad tt \mid \mathsf{type}(t_c) \mid \neg\Phi \mid \Phi \wedge \Phi \mid \texttt{<}m_c\texttt{>}\ \Phi \mid \mathbf{G}\Phi \mid \Phi\mathbf{U}\Psi$$

where t_c and m_c represent type and module constraints, respectively.

SLTL formulas are interpreted over the set of all *finite legal design plans*, i.e. all the sequences of design tasks where the output type (signature) of each task coincides with the input type (signature) of its successor. Intuitively,

- tt (meaning true) is satisfied by every plan.
- $\mathsf{type}(t_c)$ is satisfied by every design plan whose first tool satisfies the type constraint t_c.
- Negation $\neg$ and conjunction $\wedge$ are interpreted in the usual fashion.
- $\texttt{<}m_c\texttt{>}\ \Phi$ is satisfied by plans whose first module satisfies m_c and whose *continuation*[5] satisfies Φ. In particular, $\texttt{<.>}\ \Phi$ is satisfied by every plan whose continuation satisfies Φ.
- $\mathbf{G}\Phi$ requires that Φ is satisfied for every suffix of the program composition. As we are dealing with finite paths only, certain properties of this kind will never hold. This operator is useful e.g. to express global type constraints: in our setting $\mathbf{G}\neg$`bitmap` excludes from the solution all the plans that feature modules reading/generating data of type { `tiff, jpeg, gif` }
- $\Phi\mathbf{U}\Psi$ expresses that the property Φ holds for all components of the plan, until we reach a *border* position where the corresponding continuation satisfies a property Ψ. Note that $\Phi\mathbf{U}\Psi$ guarantees that the property Ψ holds eventually (strong until). This is the most interesting construct of SLTL, because it allows to specify different properties to be met by different portions of a design plan.

The last three operators are responsible for the global character of SLTL specifications. In particular, specifications can capture the *interdependencies* of neighbouring design tasks by means of the `< >` operator, and of distant design taks by means of the $\mathbf{U}$ operator.

The on-line introduction of **derived operators** allows a modular and intuitive formulation of complex properties. In fact, as can be seen in Section 5, our built-in macro

[4]The logic can be regarded as a variant of PTL [MaWo84] or a linear-time variant of CTL [EmCl82]. The predicate 'semantic' accounts for the fact that atomic propositions and actions are interpreted over their own logics.

[5]This continuation is simply the plan starting from the second module.

facility allows natural language-like specifications. As examples we introduce the dual operators

$$\begin{array}{llll}
\textit{False}: & \mathit{ff} & =_{df} & \neg tt \\
\textit{Disjunction}: & \Phi \vee \Psi & =_{df} & \neg(\neg\Phi \wedge \neg\Psi) \\
\textit{Box}: & [m_c]\Phi & =_{df} & \neg < m_c > (\neg\Phi) \\
\textit{Eventually}: & \mathbf{F}\ \Phi & =_{df} & \neg\mathbf{G}(\neg\Phi) \quad = \quad tt\ \mathbf{U}\ \Phi \\
\textit{Weak Until}: & \Phi\ \mathbf{WU}\ \Psi & =_{df} & \neg(\neg\Psi\ \mathbf{U}\ \neg(\Phi\ \vee\ \Psi))
\end{array}$$

and convenient synonyms and derived operators which occur in the examples:

$$\begin{array}{lll}
\Phi \Rightarrow \Psi & =_{df} & \neg\Phi \vee \Psi \\
\Phi, \Psi & =_{df} & \Phi \wedge \Psi \\
\textbf{use}\ \Phi & =_{df} & \mathbf{F}\Phi \\
\textbf{finally}\ \Phi & =_{df} & \mathbf{F}\ (\Phi \wedge\ [.]\mathit{ff})
\end{array}$$

We give three further examples of constraints specified in SLTL:

1. $\texttt{<VPNR>}\ \mathit{ff}$
   imposes all legal design plans to use the VPNR tool as last module[6].

2. $\mathbf{F}(< \texttt{place} > tt)$
   This SLTL formula intuitively means: "place occurs eventually", i.e. a placement algorithm has to be used now or at a later design step.

3. Specification of a liveness property:
   $\mathbf{G}\ (\ (< \texttt{place} > tt\ )\ \Rightarrow\ \mathbf{F}(< \texttt{show} > tt)\ )$
   This SLTL formula intuitively means: "Whenever place occurs, then show occurs at the same time or later".

Note that users do not need to master the basic syntax of SLTL. They can define their constraints by means of the simpler derived operators, like in the examples of the next section.

The use of a temporal logic like SLTL especially allows the construction of *incomplete* specifications. In our setting this means in particular that SLTL enables the specification of a partial order on design tasks. This possibility is quite helpful, since there often exist situations where the order in which design tasks have to be executed is not known or irrelevant. This situation is precisely captured by the partial orders.

The next two sections illustrate the central features of our approach following a user session for the development of a new, complex special-purpose design tool, which we solve by synthesizing a corresponding design plan.

5 Working with the Synthesis Component

To illustrate the 'pure' synthesis process sketched in Figure 1, we revisit the *Lassie* task presented in [DaDi91] for the illustration of a user session with Cadweld.

[6]The continuation *ff* is in fact not satisfiable by any module.

The *Lassie* Task. The task automatically creates a data path for a partially specified design. In [DaDi91] it was described as follows:

> *The Lassie task uses a design that consists only of logical components and their required connections and creates a VLSI chip with the appropriate interconnections completed. To achieve this, the Lassie task needs a design specified in the* `DIF` *language from which to* `start`*. It then determines the* `structural binding` *between logical components and actual physical realizations. Lassie then attempts a* `global place` *and* `route` *to determine where the parts of the design should be placed in the final chip. Once this is done, the actual wires are added to the design specification and the chip is transformed into its* `final` *representation (in this case,* `CIF`*).*

The remainder of this section presents the required ingredients and steps for the treatment of this task.

The Repository. The required fragment of our repository is shown in Figures 2 and 3. It has been reconstructed from the data in [DaDi91], e.g., the number and names of tools responding to a query, and has been added to our usual library of tools and algorithms for analysis and verification of distributed systems (as indicated e.g. by entries like `verification_tools` in the module taxonomy and by similar entries in the type taxonomy).

The taxonomic classification which defines groups over the atomic modules is shown in Figure 3 as it is displayed by our hypertext system, together with the card for the atom TimberWolfSC. Cards contain a concise textual description (expandable to full documentation via hypertext links), which may also include important warnings, links to examples of application and information about the collocation in the taxonomy (here, the parent groups are listed). For the atoms, the card also contains their input and output types, which serve as on-line links to the type taxonomy. In this card the types are dif and gpc, an abstract type indicating that the global placement has been completed. Taxonomies can reflect very different classification criteria: in the module taxonomy, not only the functionality of the tools has been captured (by means of predicates like `route`, `place`, `cad_tools`), but also the orthogonal robustness criterion, which plays an important role in the development of this example.

The Specification. Already the formulation of the Lassie task shown in Fig. 4 (middle) is understood by the system. The formula largely consists of the highlighted keywords contained in the plain text description above. This formula is expanded in terms of primary SLTL operators to give

$$\mathsf{type}(\mathtt{dif}) \wedge \mathbf{F}(\mathtt{bind}\wedge\mathtt{structure}) \wedge \mathbf{F}(\mathtt{place}\wedge\mathtt{global}) \wedge \mathbf{F}\,\mathtt{route} \wedge \mathbf{F}(\mathsf{type}(\mathtt{cif}) \wedge [.]\mathit{ff})$$

which is passed to the synthesis algorithm. Note the looseness of the formula: no temporal constraints have been formulated for orderings between the modules. The only precise requirements are to start with type `dif` and to terminate with type `cif`.

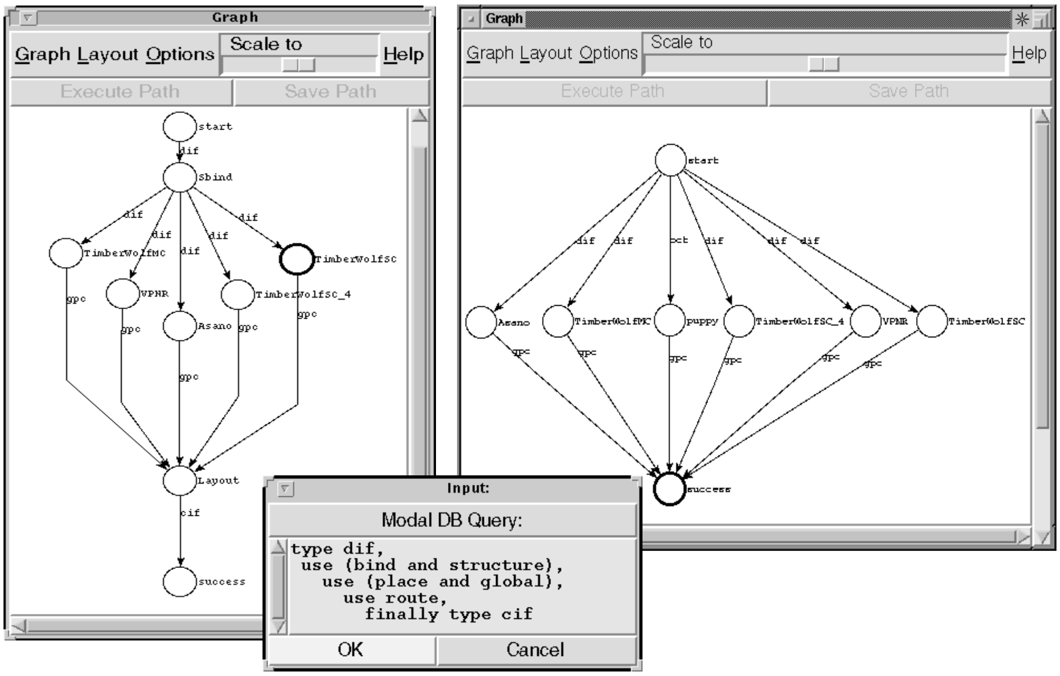

Figure 4: Synthesis from the Original Specification

The Solution Space. Given this formula, the system returns a set of minimal length design plans in graph representation: every path from the start to the success node in the graph of Figure 4(left) is a possible shortest (i.e. minimal length) solution to our synthesis problem.

In the solution graph, nodes represent modules (atoms) labelled by their name, and edges are labelled by type information. The correct sequencing of the modules along each path is ensured by type correctness.

Solutions can be investigated by means of the hypertext browser, which provides information about single modules and types as well as about the type and module taxonomies. The hypertext system offers via a mouse click a card for each node or edge of the graph.

The Execution. All the proposed solutions are directly executable. In fact, each path in the graph corresponds to a target program in our high-level programming language, which is implemented in C++. In order to invoke a solution, its corresponding path must be selected by means of the mouse. The system offers the option for execution, as soon as the selection is unambiguous. Input required during execution can either be typed in a pop-up window or loaded from a file.

In the original Cadweld session, one of these plans had been constructed "on the fly" at design time. Unfortunately, it did not meet the desires of the designer, since additional requirements (e.g. the robustness of the placement tools) were interactively discovered to play a central role too. That meant interrupting the design, and trying to backtrack stepwise to an earlier safe design stage from which the design was resumed under stricter requirements.

Exploring the Library. In our solution, looking at the graph of proposed design plans, experienced designers may know that there are still more global placement tools available in the library, and may ask METAFrame to show all of them via the local constraint

```
use global
```

This local constraint is similar to the placement of a request on the blackboard of a tool like Cadweld, or to asking for an expansion step in the course of constructing a task schema in Hercules. The tools corresponding to this description are the six shown in Figure 4(right). puppy did not appear in the previous solutions because it only accepts .oct input, which requires an additional translation step. Thus the path through puppy would be one step longer, and not minimal length. In order to consider such plans we use the option for synthesis of *minimal* rather than minimal length solutions.

Modifying the Query. In the *prototyping* phase, the user approaches the desired solution by successively refining or relaxing an initial (usually loose) specification on the basis of some hypertext investigation or experiments with automatically generated prototypes.

Most additional requirements do not really need running the tools in order to be discovered, but can be easily captured by a well-designed taxonomy and by inspection of the cards of the tools appearing in the proposed solutions. As an example, the warning in the description field relative to `TimberWolfSC` (see Fig. 3(right)) suffices to rule out any path through this tool in case CPU-friendliness is important.

In the original Cadweld session it turned out that the previous specification was still insufficient. It was necessary to start from a `.ALL` file, and additional constraints (e.g. robustness, one-dimensional placement) on the placement tool emerged. The resulting layout (`.CIF` file) was also wished to be successively inspected via `MAGIC` or plotted via `CPLOT`.

In our setting, this immediately leads to the new specification of Fig. 5. The result is shown in the same figure, this time after configuration of the synthesis algorithm in the *minimal* solutions mode. The two leftmost solutions would have been found in the shortest path mode too, however the solution through puppy is only minimal and not shortest. The three paths to the failure node, highlighted on the screen, indicate failing attempts. This indication has proved valuable for locating errors and to debug specifications without running the tools. E.g., although the module taxonomy shows that TimberWolfSC_4 is even more robust than required, it fails to satisfy the one-dimensional placement.

Note that solutions contain (necessary) modules which are not mentioned in the specification. They are automatically introduced by the synthesis algorithm as type correct (sequences of) modules which bridge gaps between different parts of the specification. As an example, we are starting now with a .ALL file, but the binder `Sbind` needs a .DIF input. A call to the `SAW_Extractor` tool is therefore appropriately introduced in the design plan as a necessary type transformer. If we made extensive use of this completion feature, already the following, much more concise specification

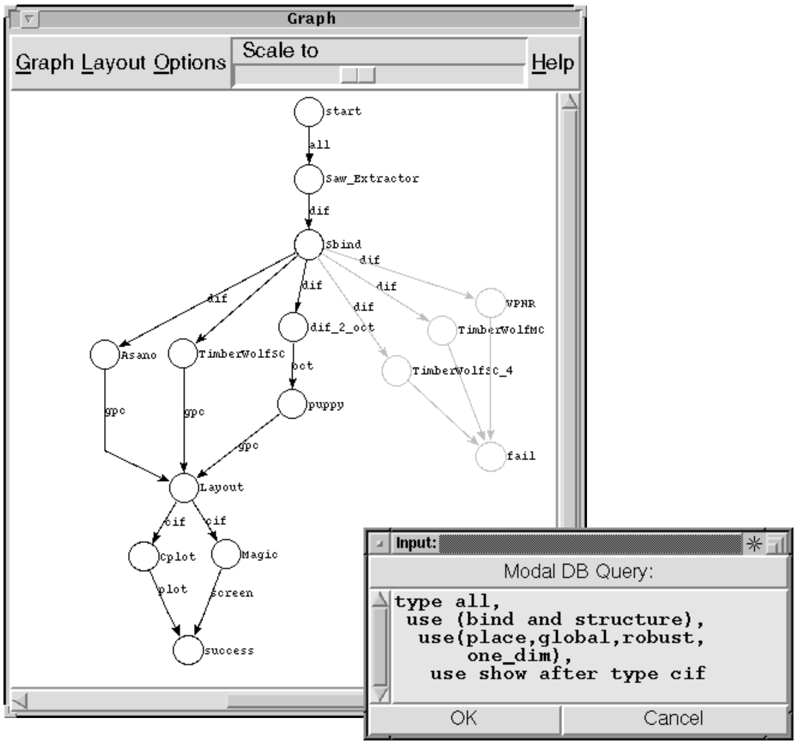

Figure 5: Synthesis with the Additional Constraints

```
type all, use (global, robust, one_dim), finally show
```

would deliver the same set of solutions in the given taxonomy.

The Compilation. Satisfactory plans can be made persistent through compilation into a new module that can be saved in the repository for later reuse. Repository updates are supported by the hypertext system by automatically associating with each new module a new card, which by default contains appropriately precomputed taxonomic criteria and a short textual description. These criteria are generated by the system from the corresponding criteria of its components. Thus the consistency of the taxonomies is automatically maintained. The user only needs to provide a new name for the new component, then the repository and taxonomies are updated automatically. Of course, the proposed content of the new hypertext card can be modified interactively too.

6 A Synthesis-Based Interactive Editor

The good performance of our global synthesis procedure is due to its restriction to linear compositions. More complex control structures glueing the linear portions together must be realized by hand, as it is the case in all the other design planning tools. Still, being able to synthesize linear plans improves drastically over previous methods, where only single components can be retrieved from the underlying repository along local criteria. The advantage of our method is already present if a designer is only interested in the functionality of single tools, because our synthesis algorithm will automatically determine (linear compositions of) required interfacing modules.

CAD-MetaFrame can be regarded as a sophisticated editor for the interactive composition of reusable design plans on the large to huge grain level, which allows the automatic synthesis of the required linear portions.
Altogether our synthesis-based editor works in a three-step rhythm:

1. decompose the problem into non-linear context components and their linear portions,
2. synthesize solutions to each linear subproblem along the lines of the previous section. In this step interaction allows to successively refine the specifications in order to obtain optimized solutions, and finally
3. construct the global implementation from the partial solutions synthesized in the previous steps.

7 Perspectives

The experience with CAD-MetaFrame is very promising: not only is it unique in making available to the users an unprecedented wealth of possibilities, but it also efficiently helps them in mastering the corresponding diversity according to specific application profiles. This feature is essential in order to encourage experimentation with the new generation of tools, which often lack acceptance by practitioners because of their high complexities.

Our approach exactly meets the description expressed by Goguen and Luqi in [GoLu95] for the emerging paradigm of *Domain Specific Formal Methods*, which requires formal methods to be introduced also on a large or huge grain level, to support programming with whole subroutines, modules and tools as elementary building blocks. This is precisely what MetaFrame is designed for. In our setting, this is reflected in the application of formal methods as a powerful aid in the coordination and combination of complex tasks, as in our synthesis of design plans which organize complex tools along formally described criteria.

Key factor to the acceptance of our methods in the industrial environment was, however, the *incremental formalization* approach [SMCB96] of MetaFrame, ranging from 'no specification', which results in the old-fashioned development style, to 'detailed specification' with full tool support. This accounts for full compatibility with existing practice, and for offering (rather than demanding) a shift in the accustomed habits of individual developers.

Acknowledgements

We are grateful to Michael von der Beeck, Volker Braun, Andreas Claßen, Achim Dannecker, Peter Faber, Carsten Friedrich, Dirk Koschützki, Falk Schreiber, Markus Schweighofer for their constructive cooperation in the design and implementation of CAD-METAFrame.

References

[BrDi95] J. Brockman, S. Director: "*The Schema-Based Approach to Workflow management*", IEEE Trans. on Computer-Aided Design Vol.14, N.10, Oct. 1995, pp. 1257-1267.

[BoPS92] D. Borrione, L. Pierre, A. Salem: "*Formal verification of VHDL descriptions in the Prevail environment*", IEEE Design & Test of Computers, June 1992.

[BuDi87] M. Bushnell, S. Director: "*ULYSSES - A knowledge based VLSI design environment*", Int. J. AI Eng., Vol.2, N.1, January 1987.

[DaDi91] J. Daniell, S. Director: "*An Object Oriented Approach to CAD Tool Control*," Trans. on Computer-Aided Design, Vol.10, N.6, June 1991, pp.698-713.

[EmCl82] A. Emerson, E. Clarke: "*Using branching time temporal logic to synthesize synchonization skeletons*," Science of Computer Programming Vol.2, pp.241-266, 1982.

[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.

[HPRS95] S. Halama, C. Pichler, G. Rieger, G. Schrom, T. Simlinger, S. Selberherr: "*VISTA – User Interface, Task Level, and Tool Integration*," IEEE Trans. on Computer-Aided Design Vol.14, N.10, Oct. 1995, pp. 1208-1222.

[JoBr95] E. Johnson, J. Brockman: "*Incorporating design schedule Management into a flow management system*", Proc. 32nd ACM/IEEE DAC, June 1995.

[KnPa86] D. Knapp, A. Parker: "*A design utility manager: the ADAM planning engine*", Proc. DAC'86, IEEE, 1986, pp. 48-54.

[MaWo84] Z. Manna, P. Wolper: "*Synthesis of Communicating Processes from Temporal Logic Specifications*," ACM TOPLAS Vol.6, N.1, Jan. 1984, pp.68-93.

[MGSW89] J. Miller, K. Groening, G. Schulz, C. White: "*The object-oriented integration methodology ogf the Cadlab workstation design environment*", Proc. ACM/IEEE 26th Design Automation Conference, 1989.

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

[SMCB95] B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß: "*An Environment for the Creation of Intelligent Network Services*", invited contribution to the book "The Advanced Intelligent Network: A Comprehensive Report", Int. Engineering Consortium Chicago (USA), Dec. 1995, pp. 287-300.

[SMCB96] B. Steffen, T. Margaria, A. Claßen, V. Braun: "*Incremental Formalization: a Key to Industrial Success* ", Appears in "SOFTWARE: Concepts and Tools", Springer, 1996.

[SMCB96b] B. Steffen, T. Margaria, A. Claßen, V. Braun: "*The* METAFrame*'95 Environment*" (Experience Report for the Industry Day) To appear in CAV'96 - Juli-Aug. 1996, New Brunswick, NJ, USA, LNCS, Springer Verlag.

[SiCl85] A.P. Sistla, E.M. Clarke: "*The Complexity of the Propositional Linear Temporal Logics*," Journal of the ACM, Vol.32, N.3, July 1985, pp.733-749.

[SNI93] "*Jessi Common Frame V2.0 - Desktop User's Guide*," Siemens Nixdorf Informationssysteme AG, 1993

[StMC95] B. Steffen, A. Claßen, T. Margaria: "*The* METAFrame*: 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.

[StMC96] B. Steffen, T. Margaria, A. Claßen: "*Heterogeneous Analysis and Verification for Distributed Systems*", "SOFTWARE: Concepts and Tools" N. 17, pp. 13-25, March 1996, Springer Verlag.

[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.

[Wagn93] F. Wagner: "*A Framework-Based Realization of an Environment for Formal Hardware Verification*," Proc. IEEE EDAC-EuroASIC'93, Paris (F), Feb. 1993, pp.41-45.

[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.

[WoBH94] P. van der Wolf, O. ten Bosch, A. van der Hoeven: "*An enhanced flow model for constraint handling in hierarchical multi-view design environments*", Proc. ICCAD'94, IEEE, Nov. 1994.