Backtracking-Free Design Planning by Automatic Synthesis in METAFrame

Tiziana Margaria Bernhard Steffen

Universität Passau* (D) Universität Dortmund** (D)

Abstract. We present an environment supporting the flexible and application-specific construction of design plans, which avoids the insurgence of unsuccessful design plans at design time, and is thus backtracking-free. 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. Example application field is the generation of design plans in a CAD environment for hardware design.

1 Motivation

To master the complexity and variety of system-level design, flexible 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. We focus here on a concrete application domain, and consider the functionality of CAD Frameworks. Themselves large and complex software systems, frameworks like e.g. NELSIS [17], CADLAB [10], or the Jessi CAD Framework [11] have succeeded to provide VLSI designers with sophisticated and user friendly project management wrt. data and design execution flow. As such, they are widely used, mainly to enforce standard design plans, i.e. predefined successions of activities (e.g. synthesis, verification, simulation, test pattern generation, and - at the physical design level - placement, routing, binding, etc...) which are carried out to get from a circuit specification through many successive design steps to the corresponding layout masks for physical realization on chips.

CAD frameworks are good at enforcing a prescribed plan of design activities, but they still offer little support for advanced, application-specific synthesis of design plans involving a *flexible use and combination* of available tools. Flexibility is however increasingly needed, since projects become larger, tool functionalities

* Fakultät für Mathematik und Informatik, Universität Passau, Innstr. 33, D-94032 Passau (Germany), tiziana@fmi.uni-passau.de

** Lehrstuhl für Programmiersysteme, Universität Dortmund, GB IV, Baroper Str. 301, D-44221 Dortmund (Germany), steffen@cs.uni-dortmund.de

more complex, and therefore standard plans need to be increasingly complemented or substituted by ad-hoc design planning on a case by case basis.

Related Work. Some flexibilization of tool management was first offered by design environments like ULYSSES [3], ADAM [8] and later Cadweld [4]. 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. Hercules [1, 7] and recent versions of NELSIS [18] enhance these methods to a workflow management which captures both tool and design management aspects. However, as a consequence of their implicit treatment of design planning, *during* design execution, all these systems detect unsuccessful design attempts only *after* some unsuccessful design step has already been concretely carried out. This requires complex and expensive backtrackings to undo wrong runtime planning decisions. Since in hardware design tool executions typically generate huge amounts of data[1], the files containing intermediate and previous results are usually destroyed after carrying out successive design steps. Therefore the amount of work wasted through backtracking is increased by the frequent need of 'undoing' many more steps, in order to start the redesign from a data consistent point. Here the frameworks' data management functionality helps in keeping track of data completeness and consistency (design plan recovery), but it would be much more efficient to avoid backtrackings at all.

Our Contribution. With CAP-METAFrame we propose the introduction of a Computer-Aided Planning phase *prior* to design execution, where design plans are generated from loose formal descriptions of the desired design tasks. Our approach to the synthesis of (linear) design plans from goal-oriented specifications is novel in its intent and its solution. Intuitively, our synthesis procedure can be regarded as a mechanism to explore the future development of plans *prior* to any execution, in a depth that allows designers to detect traps early enough to avoid backtracking. The generation of the collection of complete, executable design plans on the basis of simple but powerful constraint-like specifications (given in a temporal logic) and of the library of available CAD tools is completely automatic. Once the set of solutions is available, they can be investigated and executed. Of course this feature comes at the price of some structural restrictions on the side of the considered plans. However, linearity is no vital constraint and concurrency could be dealt with along the lines of a serialization (i.e. interleaving) semantics.

The distinctive feature of our specifications is their *global* character, i.e., in our context, their ability to express required *interdependencies* between tools (like orderings, precedences, eventualities, and conditional occurrences) when striving to solve a complex task. Exactly such interdependencies are responsible

[1] To give an idea, the now popular Binary Decision Diagrams were initially devised for use in hardware design, to cope with the huge state spaces of gate-level circuit descriptions.

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.

More concretely, constraints in the logic capture the following significant classes of *global* properties:

- general ordering properties, like *this design step must be executed some time before this other,*
- abstract liveness properties, like *a certain design step is required to be executed eventually,* and
- abstract safety properties, like *two certain design steps must never be executed simultaneously.*

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, CAP-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 CAP-METAFrame: if no formal constraints are used, local design decision are supported, however, the more constraints are used the better is the automatic support [14]. 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 [2] (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 [15] 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 which includes what shown in [4] (basically, a collection of module signatures and classification keywords), 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 [15] has been successfully used since 1995 both in-house, e.g. for the construction of heterogenous analysis and verification algorithms [13], but also in several joint projects with industrial partners (Siemens Nixdorf Munich [16, 14], Siemens AG, Telemedia-Bertelsmann, and the Springer Verlag [12]).

3 Synthesizing Plans in CAP-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 executability, tool compatibility, and other consistency conditions. Accordingly, CAP-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 a domain level formulation, 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. 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 CAP-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 shortest 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

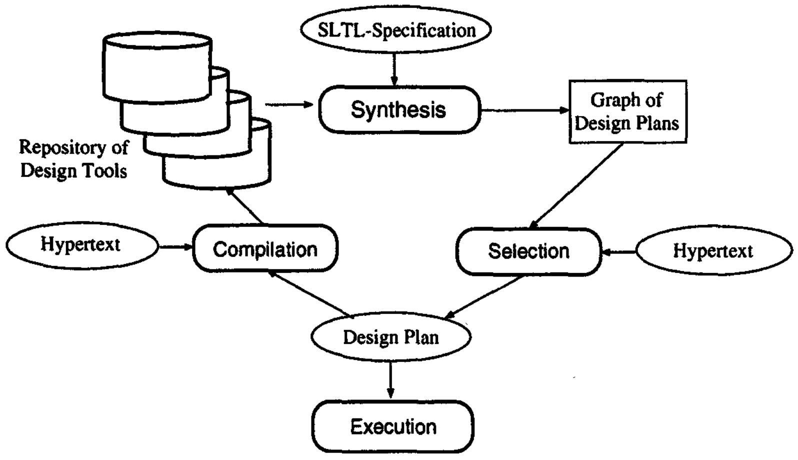


Fig. 1. The METAFrame Design Plan Synthesis Process

the current state of the tool repository, and 3) the classification scheme (called taxonomy) characterizing the application profile of each available tool.

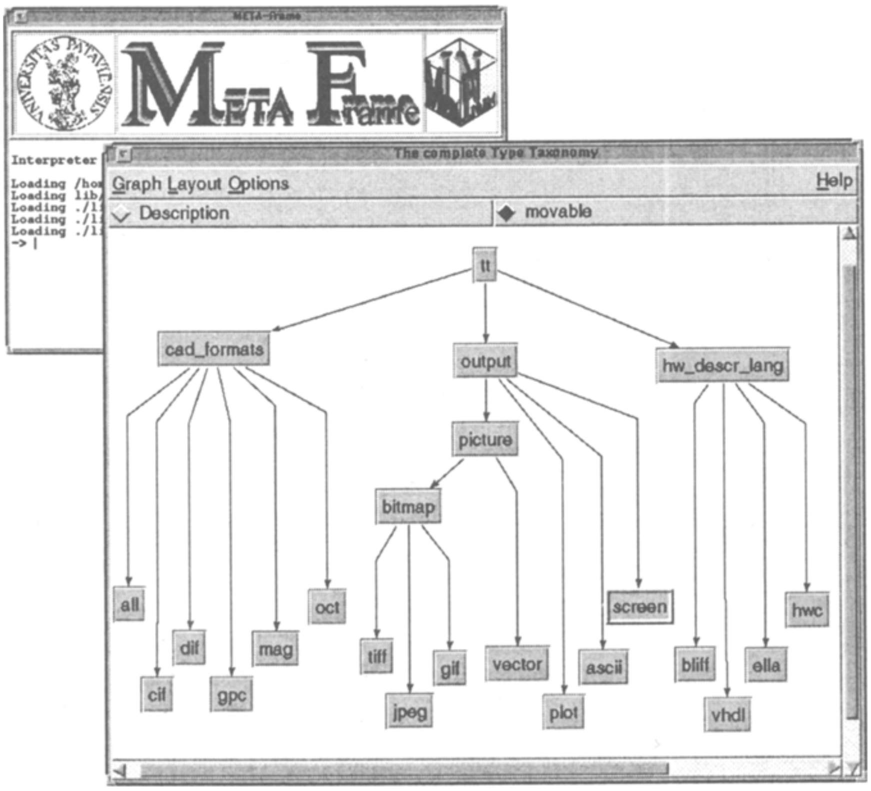
Besides proposing the set of all successful plans, our system also provides diagnostic information highlighting the reason of failure of other plans. We will illustrate the features of our tool in Section 5 along the lines of the motivating example for Cadweld [4]. This allows a direct comparison with the most similar approach. A more detailed discussion of related work can be found in [1, 13].

Our synthesis component works by model construction roughly along the lines of [9]. In general *global constraints* lead to an exponential worst-case complexity, which is not surprising for a global search problem. However, as confirmed by our experience in a joint project with Siemens Nixdorf, 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*,



```
Interpreter
Loading /ho
Loading lib
Loading ./l
Loading ./l
Loading ./l
-> |
```

Fig. 2. Main Window and Type Taxonomy

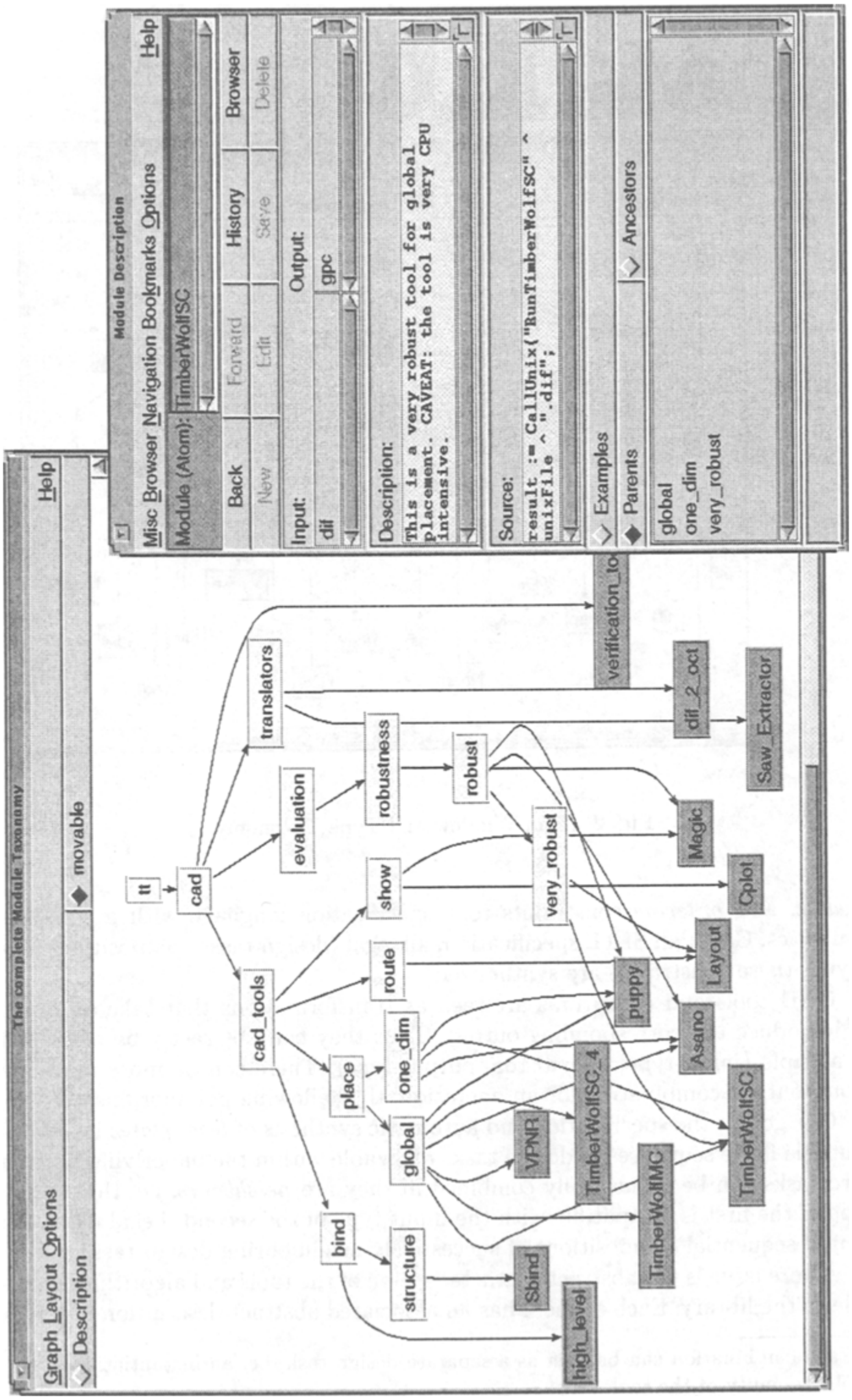
module, and *ordering* constraints to a specification language with a coherent semantics. Given an SLTL specification, all legal (design task) compositions satisfying these constraints are synthesized.

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].

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. 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

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



Description:

```
This is a very robust tool for global
placement. CAVEAT: the tool is very CPU
intensive.
```

Source:

```
result := CallUnix("RunTimberWolfSC" ^
unixFile ^ ".dif";
```

Fig. 3. Module Taxonomy and a Hypertext Card

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`

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 [1] captures exactly local constraints. Information on functional and data dependencies between tools, there represented in a task schema, is found in CAP-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 next paragraph describes the *innovative* features of our specification language, which allow automatic construction of plans from 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 extend the two local kinds of specifications to an elegant and uniform specification language for global constraints, which is defined as follows:

Definition 1 (SLTL).

The syntax of Semantic Linear-Time Temporal Logic (SLTL) is given by:

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

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

The logic can be regarded as a variant of PTL [9] or a linear-time variant of CTL [5]. The predicate 'semantic' accounts for the fact that atomic propositions and actions are interpreted over their own logics.

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.
- $<m_c>\ \Phi$ is satisfied by plans whose first module satisfies m_c and whose *continuation*[4] satisfies Φ. In particular, $<.>\ \Phi$ 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 G¬`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 a 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, since it allows to specify different properties to be met by different portions of a plan.

[3] An example for a user's local constraint query in CAP-meta is contained in Section 5.
[4] This continuation is simply the plan starting from the second module.

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 tasks by means of the **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 facility allows intuitive domain level specifications. As examples we introduce the dual operators

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

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

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

We give three further examples of constraints specified in SLTL:

1. <VPNR> *ff*
   imposes all legal design plans to use the VPNR tool as last module[5].
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.

Using a temporal logic like SLTL especially allows the formulation of *incomplete* (loose) 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] The continuation *ff* is in fact not satisfiable by any module.

5 Working with the Synthesis Component

To illustrate the 'pure' synthesis process sketched in Figure 1, we revisit the *Lassie* task presented in [4] for the illustration of a user session with Cadweld.

The Lassie *Task.* In a context of 'physical level' design of VLSI chips, this task automatically creates a data path for a partially specified design. The design is partially specified since it is given as list of (functional) components, without any information neither on the physical realization of each of them (this is contained in a library within the framework) nor on the final spatial arrangement for the final chip. These mappings are computationally very intensive, have to be carried out in several steps with several tools, and their computational efficiency as well as the quality of the resulting layout (placement of each single transistor and of the necessary wirings) may vary greatly depending on the choice among the available tools and among the algorithms offered by the tools. The ideal goal is a fast computation of a compact layout with low power consumption from the given gate-level netlist. In [4] the design task 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*[6] *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 (here,* `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 [4], 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

[6] `DIF` and `CIF` are standard hardware Interchange Formats.

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 Figure 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

$$\begin{array}{l} \mathtt{type(dif)} \wedge \\ \qquad \mathbf{F}(\mathtt{bind} \wedge \mathtt{structure}) \wedge \mathbf{F}(\mathtt{place} \wedge \mathtt{global}) \wedge \mathbf{F}\ \mathtt{route} \wedge \\ \qquad\qquad \mathbf{F}(\mathtt{type(cif)} \wedge [.]\mathit{ff}) \end{array}$$

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

The Solution Space. Given this formula, the system returns the set of shortest 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, in terms of number of tools involved) 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, only one of these plans could be constructed at a time. Moreover, this happened "on the fly", at design time. Unfortunately, the constructed plan did not meet the expectations of the designer, since additional properties (e.g. the 'robustness' of the placement tools) were interactively discovered to play a central role for a satisfactory completion of the task. That meant interrupting the design after having already run the 'wrong' placement tool, and trying to *backtrack stepwise* to an earlier (data safe and complete) stage, from which to resume the design under stricter plan requirements.

This time and resource consuming backtracking can be completely avoided *if a* global *solution space is available, which is not the case in the common CAD frameworks.*

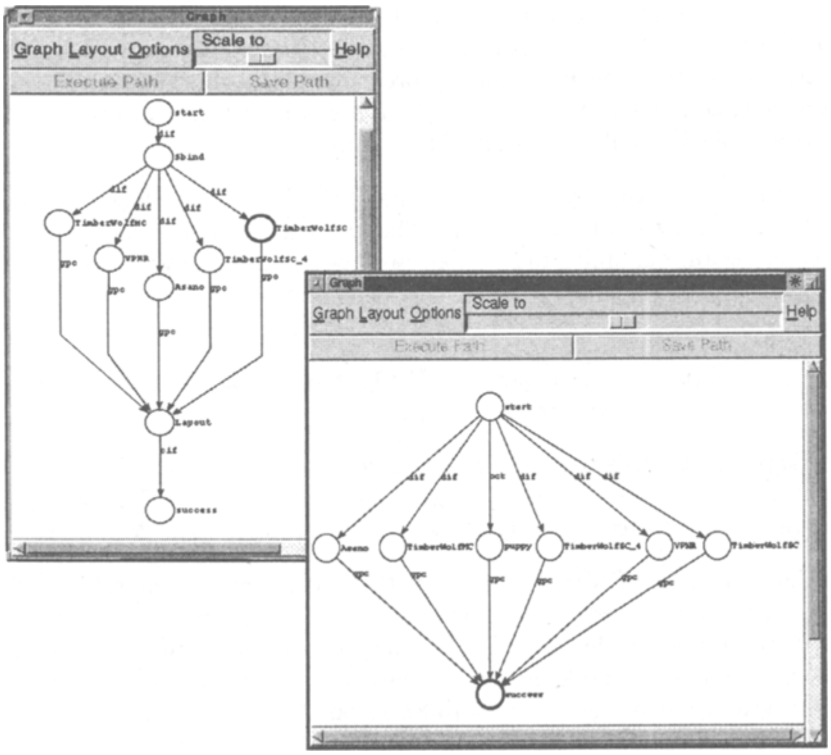


Fig. 4. Synthesis from the Original Specification

On the contrary, we have just shown that CAP-METAFrame offers the automatic generation and visualization of such a global solution space. Underspecifications are here easily detectable by inspection: domain experts may in fact notice that continuations of global plans after some operation do not meet their intuitive expectations. In this example a quick look at the proposed solutions lets a hardware designer immediately discover that some of the placement tools appearing there are not robust enough. Therefore the need for an additional requirement to the design plan is immediately detected, *prior to any execution*, and thus in particular without any run-time backtracking.

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 the .oct input format, which requires a format conversion via an additional translation step. Thus the design path through puppy would be one step longer than the ones in Figure 4(left), thus not shortest. In order to consider such plans we use the option for synthesis of *minimal*[7] rather than shortest solutions.

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

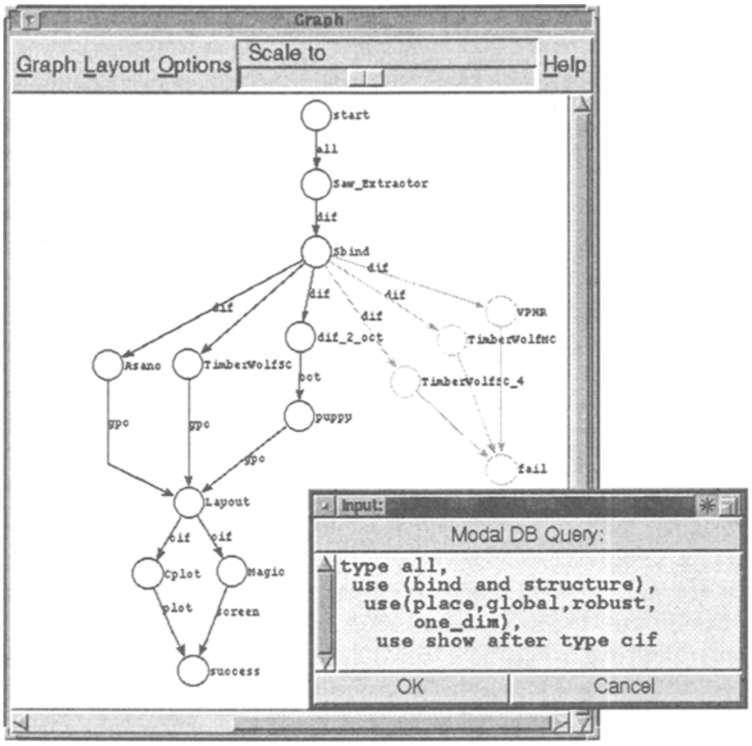
Most additional requirements do not really need running the tools in order to be discovered, but can be easily captured by a well-designed taxonomy and by inspection of the cards of the tools appearing in the proposed solutions. As an example, the warning in the description field relative to `TimberWolfSC` (see Figure 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 (1) start from a `.ALL` file, and (2) additional constraints (e.g. robustness, one-dimensional placement) on the placement tool emerged stepwise. The resulting layout (`.CIF` file) was also (3) wished to be successively inspected via `MAGIC` or plotted via `CPLOT`. The availability of global plans for inspection would have sufficed to spot the corresponding inadequacies, without requiring any run-time information.

In our setting, this domain-level knowledge immediately leads to the new specification of Figure 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 loose specification. They are automatically introduced by the synthesis algorithm as type correct (sequences of) modules which bridge gaps between different parts of the specification. Here the globality of the constraints comes into play. 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

[7] Minimal means that no sub-plan is a solution. It means in particular also cycle-free.



```
type all,
  use (bind and structure),
   use(place,global,robust,
       one_dim),
    use show after type cif
```

Fig. 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.

CAP-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 CAP-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 [6] 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 [14] 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.

The same principles and algorithms underlying the design planning facility of CAP-METAFrame are now being used as tool coordination engine in the Electronic Tool Integration platform (ETI) of the Springer Journal on *Software Tools for Technology Transfer* [12].

Acknowledgement We are grateful to Andreas Claßen for his constructive cooperation in the design and implementation of CAP-METAFrame.

References

1. J. Brockman, S. Director: "*The Schema-Based Approach to Workflow management*", IEEE Trans. on Comp.-Aided Design Vol.14(10), Oct. 1995, pp. 1257-1267.
2. M. von der Beeck, T. Margaria, B. Steffen: "*A Formal Requirements Engineering Method Combining Specification, Synthesis, and Verification,*" Proc. IEEE SEE'97, Cottbus (D), April 1997, IEEE Comp. Soc. Press, pp.131-144.
3. M. Bushnell, S. Director: "*ULYSSES - A knowledge based VLSI design environment*", Int. J. AI Eng., Vol.2, N.1, January 1987.
4. 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.
5. A. Emerson, E. Clarke: "*Using branching time temporal logic to synthesize synchonization skeletons,*" Sc. of Computer Progr., Vol.2, pp.241-266, 1982.
6. J.A. Goguen, Luqi: "*Formal Methods and Social Context in Software Development,*" Proc. TAPSOFT'95, Aarhus (DK), May 1995, LNCS N.915, pp.62-81.
7. E. Johnson, J. Brockman: "*Incorporating design schedule Management into a flow management system*", Proc. 32nd ACM/IEEE DAC, June 1995.
8. D. Knapp, A. Parker: "*A design utility manager: the ADAM planning engine*", Proc. DAC'86, IEEE, 1986, pp. 48-54.
9. Z. Manna, P. Wolper: "*Synthesis of Communicating Processes from Temporal Logic Specifications,*" ACM TOPLAS Vol.6, N.1, Jan. 1984, pp.68-93.
10. 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.
11. "*Jessi Common Frame V2.0 - Desktop User's Guide,*" SNI AG, 1993.
12. B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration Platform: Concepts and Design,* Int. Jour. on Softw. Tools for Techn. Transfer, Vol.1, Springer V., Dec. 1997. See `http://eti.cs.uni-dortmund.de` for informations on this service.
13. B. Steffen, T. Margaria, A. Claßen: "*Heterogeneous Analysis and Verification for Distributed Systems*", Software: Concepts and Tools 17(1), pp.13-25, March 1996, Springer Verl.
14. B. Steffen, T. Margaria, A. Claßen, V. Braun: "*Incremental Formalization: a Key to Industrial Success*", Software: Concepts and Tools 17(2), pp.78-91, Springer V., July 1996.
15. B. Steffen, T. Margaria, A. Claßen, V. Braun: "*The* METAFrame*'95 Environment*", Proc. CAV'96, Aug. 1996, New Brunswick, NJ, USA, LNCS, Springer Verlag.
16. B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß: "*An Environment for the Creation of Intelligent Network Services*", in "The Advanced Intelligent Network: A Comprehensive Report", Int. Engin. Consort., Chicago, Dec. 1995.
17. P. van der Wolf, P. Bingley, P. Dewilde: "*On the Architecture of a CAD Framework: The NELSIS Approach,*" Proc. EDAC'90, IEEE Comp. Soc. Press, pp.29-33.
18. 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.