# Restricted Design Spaces: Visualization and Consistency Tools

Vladimir Támara Patiño
tamara@informatik.uni-kl.de

03/01

Sonderforschungsbereich 501


February 2001
Universität Kaiserslautern, Kaiserslautern, Germany

## Abstract

In this article a restricted notion of design space is introduced to model variability in a system by controlling variations on the source code. Along with the definition, we propose a graphical notation, a XML file format and translation schemes to existing tools. We present also details of an implementation of several tools developed or adapted to the introduced notion, as well as a complete example of the usage of the notion and the tools: the configuration of a toy kernel for embedded controllers. Compared with related work, the main features of the restricted notion of design space here introduced are: (1) the restricted definition doesn't support numeric ranges as source of variability (2) it has associated a property of consistency and (3) there is a decidable procedure to check the consistency of a restricted design space during its construction or during its usage.

# Contents

# List of Figures

# Acknowledgments

I would like to thank the Deutsche Forschungsgemeinschaft Bonn for the research grants given to the project SFB 501 of the Universität Kaiserslautern, since this work has been sponsored by such project. I would like to thank specially to the professors directly involved with it: Jürgen Nehmer and Klaus Madlener.

I also would like to thank the persons, who have contributed with suggestions for this article or parts of it: Gustavo Ospina, Prof. Klaus Madlener, Melissa Giraldo.

CHAPTER 1

# Introduction

The idea of **design spaces** as a technique to model variability in user graphical interfaces was introduced in [**21**]. This idea was extended in [**9**] to organize requirements for product lines of embedded operating systems, and it has been used in several works and tools of the groups B5/B10 in the SFB501 project ([**10**], [**8**], [**4**], [**5**], [**2**], [**6**], [**3**],[**7**]).

Based on these works, we have worked on a restricted definition of design space that allows the checking of its consistency. As the developed definition is restricted, it has allowed to define a graphical notation as well as an XML format appropriate to inter-operate with existing tools. To validate the concept we have developed prototype tools to:

- support the visualization of a restricted design space with the graphical notation,
- check the consistency of the restricted design space,
- inter-operate with existing tools (Reboost and CML2),
- configure the sources of a toy kernel for embedded systems.

The rest of this article is organized as follows: in the section 2 we present an intuitive notion of restricted design space, in the section 3 we present a formalization of the notions given in section 2 and we present the associated property of consistency, in the section 4 we present a graphical notation, in the section 5 a XML file format is introduced, in the section 6 some details of the prototype tools are presented and finally in the section 7 a complete example is shown.

CHAPTER 2

# Informal description

The informal notion of restricted design space presented here, allows to model variability in the sources of a program, it has been developed along with the formal definition based on the related works, trying to preserve the notation and some fundamental ideas.

In a restricted design space each individual variability is called an **option**[1] and several mutually exclusive options can be grouped in a **dimension**. Other kind of restrictions between the options or the dimensions can be expressed with **correlations** and **hierarchies**. To facilitate the construction and usage of big design spaces, several dimensions and correlations can be put together in **groups**. The act of selecting individual options of a design space, in accordance to the restrictions is called **profiling**, and during profiling of a huge design space the order in which groups and dimensions are presented to the user can be organized in a **sequence**.

These are more elaborated descriptions of some of the introduced terms:

- During profiling each option can be selected, its selection can be prohibited or it cannot be selected nor prohibited.
- Dimensions are composed of mutually exclusive options. Generally when a design space is profiled exactly one option of each dimension must be selected (the exception could be certain dimensions hidden by hierarchies as we explain below).
- Correlations are logical restrictions between options. To express these restrictions, a correlation has certain **type** as well as a **left boolean expression** and a **right boolean expression**. In these boolean expressions each option is considered a boolean variable (*true* if selected, *false* if prohibited) that can be connected with the operators "$\neg$", "$\wedge$" and "$\vee$" with the usual rules to form boolean expressions. When a design space is profiled these expressions can evaluate to *true* or to *false* (with the usual meaning of the connectives and the usual procedure to evaluate a boolean expression). The possible types for a correlation, are defined in terms of the evaluation of its boolean expressions:
  - **Symmetric Positive**: Both boolean expressions of the correlation must be *true* or none of them.
  - **Symmetric Negative**: Exactly one of the two expressions must be *true*.

[1] We have tried to preserve the terms used in the previous works, specifically the terms introduced in [**9**, Rothkugel97]. In that writing the term **category** was used to express atomic variabilites; we think that the word category doesn't give the notion of atomicity, hence in this writing we prefer to call **option** to each atomic variability.

  - **Antisymmetric Positive**: If the left expression is *true* then the second also must be *true*.
  - **Antisymmetric Negative**: If the first expression is *true* then the second should be *false*.
- A hierarchy can hide or show one dimension based on the value of a left boolean expression. If the expression is true the dimension is visible, but the entire dimension will be hidden if the expression is false.
- A group contains several dimensions, and each dimension can belong at most to one group.
- A sequence is an organization of the dimensions as nodes of a forest. This forest can be used to profile the restricted design space in a certain order.

CHAPTER 3

# Formal definition

The objective of our definition is to encode some of the given notions in a well known mathematical framework, where the consistency can be checked. By doing it, we have available the full power of the framework with all the tools that other people have developed for it (i.e theorems and software). We think that the framework of mathematical logic is a very natural one for this problem, but the precise selection of which logic to use is crucial since a wrong selection could not ensure decidability of the consistency —which is our main objective. For example it could happen if we use full first order logic or full temporal logic. To ensure the objective, we have selected a very simple and also very well known logic: *propositional logic*.

What follows is the formal definition, that probably is difficult to understand in a first reading, however the explanation that follows it and the examples given below probably will help (the notation and the main ideas of propositional logic here used follow [**13**]).

Definition 3.0.1. Given a set of sentential symbols $L$, a **restricted design space** is a set of sentences $\Sigma$ where each sentence has sentential symbols in $L$ and its syntax has one of the following forms:

1. A sentence of the form
$$ExactlyOne(v_1, v_2, ..., v_n)$$
where $v_i$ is a sentence symbol that doesn't appear in any other sentence of form 1. $ExactlyOne(v_1, v_2, ..., v_n)$ abbreviates the proposition
$$(v_1 \wedge \neg v_2 \wedge ... \neg v_n) \vee (\neg v_1 \wedge v_2 \wedge ... \neg v_n) \vee ... (\neg v_1 \wedge \neg v_2 \wedge ... v_n)$$
2. A sentence of the form
$$(e \rightarrow ExactlyOne\,(v_1, v_2, ..., v_n)) \wedge (\neg e \rightarrow (\neg v_1 \wedge \neg v_2 \wedge ... \wedge \neg v_n))$$
where $e$ is an expression which includes the connectors "$\neg$", "$\wedge$", "$\vee$" and only sentence symbols that appear in other propositions of form 1 or 2 and that do not belong to the set $\{v_1, v_2, ..., v_n\}$. $v_1, v_2, ..., v_n$ should be sentence symbols that don't appear in any sentence of form 1. Any of them could appear in the expression $e'$ of other sentence of form 2
$$(e' \rightarrow ExactlyOne\,(v'_1, v'_2, ..., v'_m)) \wedge (\neg e' \rightarrow (\neg v'_1 \wedge \neg v'_2 \wedge ... \wedge \neg v'_m))$$
but if any of them appear in the part $ExactlyOne\,(v'_1, v'_2, ..., v'_m)$ then $m$ must be equal to $n$, and $v'_1, v'_2, ..., v'_m$ must be a permutation of $v_1, v_2, ..., v_n$.
3. A sentence symbol. The symbol must appear in exactly one sentence of form 1 or in some sentences of form 2.
4. The negation of a sentence symbol. The symbol must appear in a sentence of the form 1 or in some sentences of the form 2.

5. A sentence of one of the following forms: (a) $e_1 \leftrightarrow e_2$ (b) $e_1 \oplus e_2$ where $\oplus$ represents "exclusive or", (c) $e_1 \rightarrow e_2$, (d) $e_1 \rightarrow \neg e_2$. Where $e_1$ and $e_2$ are sentences that include only the connectors "$\neg$", "$\wedge$", "$\vee$" and sentence symbols that appear in sentences of form 1 or 2.

The options are sentential symbols, a selected option has the form 3, a prohibited option is a negated variable i.e. form 4. A dimensions not associated with any hierarchy has the form 1, a hierarchy along with the dimension that it can show/hide has the form 2 —note that the definition of hierarchy allows that several hierarchies show/hide the same dimension. Finally a correlation must have the form 5 in one of these ways: 5.a for a symmetric positive correlation, 5.b for a symmetric negative correlation, 5.c for an antisymmetric positive correlation and 5.d for an antisymmetric negative correlation.

Note that in this definition we have omitted the concepts of groups and sequences, we think that with the given informal description both concepts are not crucial for the consistency property, they just help to "visualize" or use big restricted design spaces.

## 3.1. Examples

The first example of restricted design space, uses the following set of sentence symbols (i.e options)

$$L = \{A.x, A.y, A.z, B.m, B.z, C.q, C.p\}$$

The sentences are:

$(C.q \rightarrow \mathit{ExactlyOne}\,(A.x, A.y, A.z)) \wedge (\neg C.q \rightarrow (\neg A.x \wedge \neg A.y \wedge \neg A.z))$
$\neg A.y$
$A.x$
$\mathit{ExactlyOne}\,(B.m, B.z)$
$B.m$
$\mathit{ExactlyOne}\,(C.p, C.q)$
$(A.x \wedge A.y) \leftrightarrow B.m$
$B.z \rightarrow \neg\,(C.p \vee C.q)$

Note that this design space has three dimensions, let's call them $A$, $B$ and $C$. The dimension $A$ has the options $A.x$, $A.y$ and $A.z$, $B$ has the options $B.m$ and $B.z$, while $C$ has the options $C.p$ and $C.q$. The options $A.x$, and $B.m$ are selected while $A.y$ is prohibited (nothing is said about the other options). There is a hierarchy that "shows" the dimension $A$ when the option $C.q$ is selected. Finally there are two correlations: $(A.x \wedge A.y) \leftrightarrow B.m$ which is a positive symmetric correlation and $B.z \rightarrow \neg\,(C.p \vee C.q)$ which is a negative asymmetric correlation .

The second example is a piece of the complete example that will be shown in the section 7. In the implementation of the kernel described there, locks were built with semaphores, while monitors were implemented with semaphores and locks. We could model both situations in a restricted design space with hierarchies, for example with the language

$$L = \{Sem.Yes, Sem.No, Lock.Yes, Lock.No, Mon.Yes, Mon.No\}$$

and the sentences:
$\mathit{ExactlyOne}\,(Sem.Yes, Sem.No)$
$\mathit{ExactlyOne}\,(Mon.Yes, Mon.No)$

$ExactlyOne\,(Lock.Yes, Lock.No)$
$(Sem.Yes \rightarrow ExactlyOne\,(Lock.Yes, Lock.No)) \wedge$
$(\neg Sem.No \rightarrow (\neg Lock.Yes \wedge \neg Lock.No))$
$(Sem.Yes \wedge Lock.Yes \rightarrow ExactlyOne\,(Lock.Yes, Lock.No)) \wedge$
$(\neg\,(Sem.No \wedge Lock.Yes) \rightarrow (\neg Lock.Yes \wedge \neg Lock.No))$

The third example is another piece of the toy kernel, it states with a correlation a restriction between tracing and assertions. The assertions can be activated or deactivated, the tracing level can be high, medium, low or non existing, but if there is no tracing there cannot be assertions. The language is

$$L = \{Asrt.Yes, Asrt.No, Tr.low, Tr.medium, Tr, high, Tr.none\}$$

the sentences are:
$ExactlyOne\,(Asrt.Yes, Asrt.No)$
$ExactlyOne\,(Tr.high, Tr.medium, Tr.low, Tr.none)$
$(Tr.none \rightarrow Asrt.No)$

## 3.2. Consistency

Intuitively a design space is consistent if a user can select exactly one option in each available dimensions (i.e. in the dimensions not hidden by hierarchies), while all the correlations are satisfied.

This notion translated to predicate logic is equivalent to require the existence of a model. Informally a model is a "world" where the sentence symbols are true or false and where all the sentences are satisfied.

Definition 3.2.1. Given a set of sentential symbols $L$ for a restricted design space $\Sigma$, the design space is **consistent** if exists a model $A \subseteq L$ for $\Sigma$ i.e $A \models \Sigma$

In propositional logic, a set of premises that has a model is called **satisfiable** while the notion consistency is related with deductions that can be inferred from the sentences (in propositional logic, a set of sentences —premises— is consistent if exists a sentence that can not be inferred from the set of premises [**13**]).

However, the completeness theorem of propositional logic ensures that both notions are equivalent (i.e the notions of satisfiable and consistent), and hence our definition of consistency is equivalent to the usual notion in propositional logic.

A big advantage that we have with this identification is that we have available the tools developed to check satisfiability of a set of premises in propositional logic, and in fact there are several methods to check this (truth tables, binary decision diagrams, resolution, several calculus for proofs, etc.), and there are also computational tools (e.g theorem provers) that implement some of these methods.

As an example the inconsistency of the first example of design space presented is shown by using only the sentences as premises, tautologies and Modus Ponens as rule of inference:

1. $\neg A.y$ is premise.
2. $A.x$ is premise.
3. $B.m$ is premise.
4. $(A.x \wedge A.y) \leftrightarrow B.m$ is premise.
5. $((A.x \wedge A.y) \leftrightarrow B.m) \rightarrow (B.m \rightarrow (A.x \wedge A.y))$ is a tautology.
6. $B.m \rightarrow (A.x \wedge A.y)$ by modus ponens with 5 and 4.
7. $(A.x \wedge A.y)$ by modus ponens with 6 and 3.

8. $(A.x \wedge A.y) \rightarrow A.y$ is a tautology.
9. $A.y$ by modus ponens with 8 and 7.

And in 9 we have a contradiction with 1.

This proof simply shows that the correlation $(A.x \wedge A.y) \leftrightarrow B.m$ is not satisfied. However the first example has something more interesting: there is not any possible selection that satisfies all the sentences! Even if all the options are neither selected nor prohibited it is inconsistent.

By the contrary the second example has models, for example one model is:

$\{Sem.yes, Mon.Yes, Lock.Yes\}$

or in other words selecting semaphores, locks and monitors make all the sentences true.

## 3.3. Consequences

Our definitions of restricted design space and consistency have several consequences, in this section we present some of them. The first one is the equivalence of a restricted design space with full propositional logic, which shows that our restricted concept is expressive enough.

Proposition 3.3.1. *The expressiveness of a restricted design space is the expressiveness of the full propositional logic. i.e. any set of sentences of propositional logic can be represented with a restricted design space.*

Proof. Given any set of premises in propositional logic:

1. Transform all the premises to CNF (Conjunctive Normal Formal) i.e to a conjunction of disjunctions. Then transform each disjunction to a rule of the form $l_1 \wedge l_2 \wedge ... l_n \rightarrow r_1 \vee r_2 \vee ... r_m$, where each $l_i$ and $r_j$ are sentential symbols. This transformation can be done always, see the details for example in [**12**].
2. Begin the restricted design space by making each sentential symbol as a dimension with one option. i.e $ExactlyOne(v)$ for each variable $v$. Then transform the rules in the following way:
   (a) For rules of the form $l_1 \wedge l_2 \wedge ... l_n \rightarrow r_1 \vee r_2 \vee ... r_m$, with $n > 0$ and $m > 0$, write the rules as positive antisymmetric correlations. The left side of the rule becomes the left side of the correlation and the analogous occurs with the right side. As each side of a rule is composed only of "or" connectives (for the right side), "and" connectives (for the left side) and sentential symbols these sides can be represented with the syntax for expressions for restricted design spaces
   (b) For rules of the form $l_1 \wedge l_2 \wedge ... l_n \rightarrow$, as each $l_i$ is a sentential symbol write $n$ sentences: $l_1$, $l_2$, ..., $l_n$.
   (c) For rules of the form $\rightarrow r_1 \vee r_2 \vee ... r_m$, as each $r_i$ is a sentential symbol write $m$ sentences: $\neg r_1$, $\neg r_2$, ..., $\neg r_m$

□

Regarding the consistency we have an important consequence that ensures that we satisfy the objective of this work: decidability.

Corollary 3.3.2. *Checking the consistency of a restricted design space is a decidable problem that is NP-complete.*

Proof. The problem of finding a model for a set of premises in predicate logic is known as the SAT problem or the satisfiability problem. By the proposition

3.3.1, deciding the consistency of a restricted design space is equivalent to the SAT problem.

Since it is decidable, there exists several procedures to solve the SAT problem, however, the SAT problem was the first problem to be shown NP-complete and hence checking the consistency of a restricted design space is also a NP-complete problem. □

The corollary 3.3.2 gives a theoretical good result for this work, but shows a big practical limitation. Note that if we choose a more powerful framework the situation will be worst [**1**]:

| Framework | Decidable | Inherent Complexity |
|---|---|---|
| Propositional Logic | Yes | $2^{cn}$ |
| Presburger Arithmetic | Yes | $2^{2^{cn}}$ |
| First order Logic | No | |
| Temporal Logic | No | |

Then, to make a definition of design space along with a decidable consistency checker that runs in polynomial time, we would have to restrict even more the definition. A relevant question would be: which additional restrictions should we impose to the presented concept in order to retain enough expressivity to model variability of sources and maintain decidability but achieve it with an algorithm which runs in polynomial time?.

For example if we could restrict our framework to deal only with Horn clauses or with quadratic Horn clauses, we would have decidability and there are algorithms that implement it with complexity $O(n)$.

However, although we could restrict the correlations of a design space to be Horn clauses or quadratic Horn clauses, we would have a problem with dimensions and hierarchies, since the sentences that define them cannot be expressed neither with Horn clauses nor with quadratic Horn clauses.

CHAPTER 4

# Graphical Notation

A good graphical notation for our notion would save us a lot of time during the construction and profiling of a restricted design space. We have developed a graphical representation that tries to be intuitive, and that can encode in a 1-1 way any restricted design space. To introduce it we show the first example of the section 3.1:

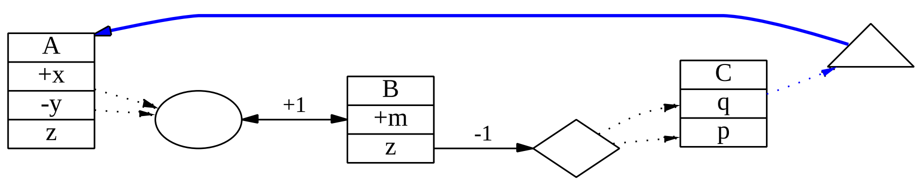


FIGURE 4.0.1. Graphical representation of a design space.

In this graph, the three dimensions are the three records $A$, $B$ and $C$, whose options are respectively $(A.x, A.y, A.z)$, $(B.m, B.z)$ and $(C.q, C.p)$, note that the names of the options don't need to include the dimension. To distinguish the selected options $A.x$ it has a plus sign while the prohibited option $A.y$ is distinguished with a minus sign. There is a hierarchy represented with a thick edge, the expression of this hierarchy begins in $C.q$ and goes through a negation (the triangle), the hierarchy hides/shows the dimension $A$. A positive symmetric correlation, relates $A.x \wedge A.y$ with $B.m$ (the oval is the connector "$\wedge$"). A negative asymmetric correlation relates $B.z$ with $C.q \vee C.p$ (the diamond is the connector "$\vee$").

In this notation, each restricted design space has been identified with an unique decorated directed graph, whose nodes can be:

- records, to represent dimensions or
- figures (ovals, diamonds and triangles), representing expressions ("and", "or" and "not" respectively)

Arcs can be dashed, thin or thick. Dashed arcs connect expressions; thick arcs always must end in records and they represent hierarchies; thin arrows represent correlations between two expressions.

Each dimension is a record whose fields are the options. The dimension is decorated with its name and each field is decorated with the name of the option. A selected option has a "+" sign preceding the name, while a prohibited option has a "-" sign preceding the name. If an option is neither selected nor prohibited it is preceded with a blank space.

An expression is represented with its syntax binary tree, the arcs must be dashed lines and the nodes can be ovals ($\wedge$), diamonds ($\vee$) or triangles ($\neg$), the leaves must be options. A left expression is represented with the syntax binary tree of the expression, but the arcs go from each son to its parent (they go from leaves to root). A right expression is the syntax binary tree of the corresponding expression with arcs going from father to son (they go from root to leaves).

A correlation is represented as a thin arc between the roots of the left expression and the root of a right expression (note that if any of the expressions is just an option, the arc of the correlation can begin in the option if it is the left expression, or it can end in the option if it is the right expression). These arcs should be decorated with "+1" or "-1" to represent positive or negative correlations, for antisymmetric correlations the arc must begin in the left expression and end in the right one i.e the arrow of the arc points the root of the right expression. For symmetric correlations the arc has arrows at both sides.

A hierarchy is represented as a thick arc from the root node of its left expression to the dimension of the hierarchy.

Groups can be represented as clusters of nodes or with colors, while sequences could decide the layout of the graph.

CHAPTER 5

# Restricted Design Spaces in XML

To allow interchanging information and to facilitate the adoption of other tools, we have developed a XML file format. The DTD that defines the grammar of this format is in Appendix A. What follows is the description of the first example of section 7 in the developed XML file format:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dsx SYSTEM "dsx.dtd">
<dsx ver0_3="doc1">
<dimension id="A">A
<option id="x" selected="true">x</option>
<option id="y" selected="false">y</option>
<option id="z">z</option>
</dimension>
<dimension id="B" group_id="2">B
<option id="m" selected="true">m</option>
<option id="z">z</option>
</dimension>
<dimension id="C" group_id="2">C
<option id="q">q</option>
<option id="p">p</option>
</dimension>
<correlation symmetric="true" positive="true">
<expr><and> <expr id_opt="x"/> <expr id_opt="y"/>
</and></expr> <expr id_opt="B.m"/> </correlation>
<correlation symmetric="false" positive="false"
description=""> <expr id_opt="B.z"/> <expr><or> <expr
id_opt="C.p"/> <expr id_opt="C.q"/> </or></expr>
</correlation>
<hierarchy idDimension="A" description=""> <expr><not> <expr
idCategory="C.q"/> </not></expr> </hierarchy>
<seq_dim id_dim="A">
<seq_dim id_dim="B"/><seq_dim id_dim="C"/>
</seq>
</dsx>
```

Some things to highlight of this file format:

- The names of the categories in expressions can be preceded with the name of the dimension and a dot, although it is not necessary if the name is unique.
- The optional attribute `selected` of an option can be only `true` or `false`, to indicate selected or prohibited options.
- The attributes positive and symmetric of the correlations should be only `true` or `false`.

- The attribute `idDimension` of the tag `hierarchy` indicates the dimension that the hierarchy hides/shows.
- Without considering groups and sequences, the translation to and from predicate logic is direct. In the translation to logic, the name of each option should always begin with the name of the dimension.
- The identifications of the options can be used internally and the descriptions can be used in the graph as decorations. The same could be said about dimensions and hierarchies.
- Each dimension can belong to at most one group, to the group identified with the optional attribute `id_group`.
- The expressions that have the attribute `id_opt` should not have sons, the expressions that don't have that attribute should have sons. The value of the attribute `id_opt` should be the identification of an option.
- The sequence is a forest, where each dimension should be exactly in one node.

CHAPTER 6

# The prototype tools

To support the notion introduced we have developed some tools, to support:

- Visualization of the graph of a design space.
- Checking the consistency of small design spaces.
- Interaction with the existing toy kernel GCRTK to configure it.
- Interaction with a existing tools developed in the context of the SFB 501 project, tools that support a broader notion of design space.
- Interaction with a different tool, developed outside the SFB 501 project but with a similar objective: CML2 that is being proposed to configure the kernel of the operating system GNU/Linux.

In order to have a functional prototype soon, we have used existing components that solve partially the required functionality and we have created translators from the XML format to appropriate formats for the reused tools.

The following graph tries to show the state of implementation in the moment of this writing:

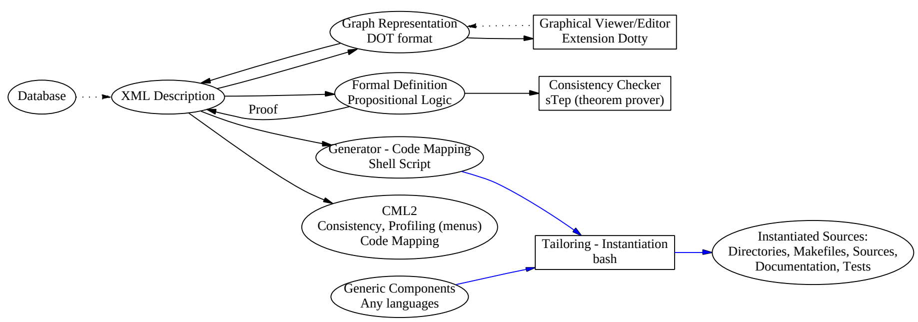


FIGURE 6.0.1. State of implementation of the prototype tools.

Ovals correspond to data, boxes are programs, edges between ovals are translators. Dashed lines indicate translators that have not been finished or that cannot be completed.

In the rest of this section, we present some details about the selection of the tools, the implementation of the translators, the integration and some possible enhancements. The technique to customize the sources of the toy kernel will be presented in detail in section 7.

## 6.1. Visualization

To visualize the graph of a restricted design space, we have used an existing package to plot graphs: Graphviz [**16**] [**15**]. We developed a first prototype on top of Graphviz (more precesily an extension to the program "dotty" in the language "lefty"), that allows the visualization of a restricted design space and also gives some capabilities of edition (particularly selecting or prohibiting an option). Our extension also allows to save in the XML file format, call the tool to check the consistency, call the tool to generate CML2 code, and call the tool that allows configuration of sources.

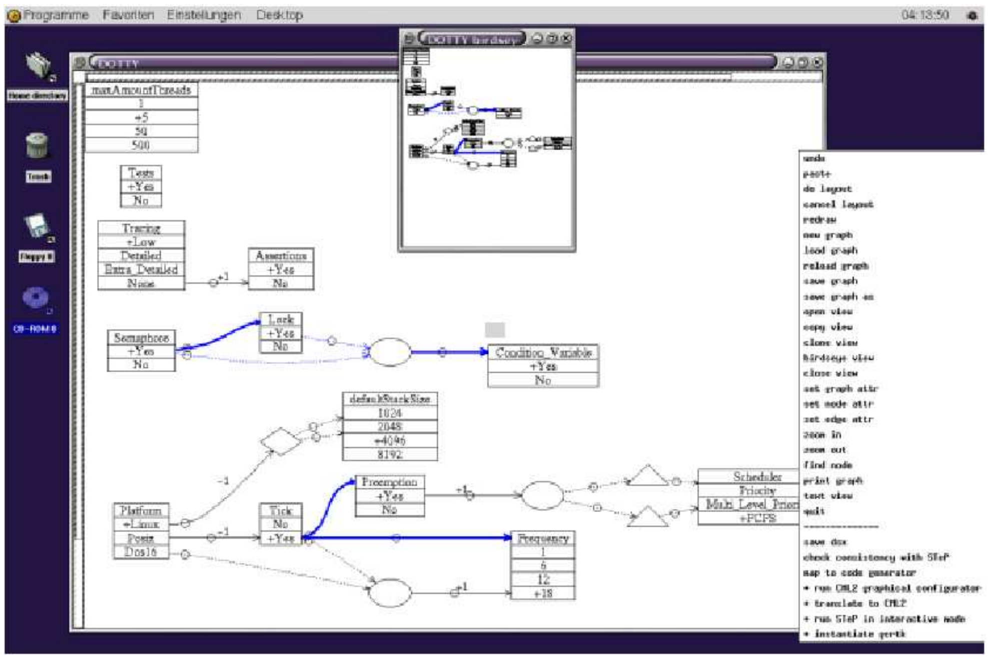


FIGURE 6.1.1. Screen-shot of visualization tool running under GNU/Linux.

To integrate this tool with the rest, we wrote two translators in the language Ocaml [**17**] by using the library PXP [**20**]. One to translate from the input file format of these tools (specifically the `dot` file format) to the XML file format, and the other to translate from XML to `dot`. All should be portable to any platform running Ocaml 3.0, although we have test only under GNU/Linux.

One advantage of dotty (the program to draw graphs of Graphviz) is its ability to calculate a good layout for the graph, in this way the user can concentrate in the restricted design space and not in its appearance. In the current implementation all the components of the graphical notation have been included, except groups and sequences.

We consider that future enhancements in the visualization tools include:

- Change to a different tool, since the copyright terms of Graphviz impose restrictions in the redistribution of derived works.
- The existing viewer could help to contract and expand some parts to give the idea of abstraction and facilitate the visualization of big design spaces.
  - A big expression tree could be contracted in a node (all the options involved in the expression could be connected to it with arcs).

  – A group, could be contracted in a node (all the arcs that connect nodes outside the group with elements of it could go to the contracted node).
- Although the existing viewer has a minimal functionality of editor, it could evolve to support the edition of any component of the XML format (for example groups, sequences and commentaries).Implement groups with clusters of nodes or with colors.

### 6.2. Consistency Checker

There are several methods to decide if a set of sentences is satisfiable, and several of them have been already implemented in theorem provers. To construct a first prototype, we have used the Standford Temporal Theorem Prover STeP [**11**]. This theorem prover uses a method called Binary Decision Diagrams, that is efficient compared with other methods that decide the existence of a model in predicate logic.

To integrate it with the other tools we wrote a translator from the XML file format to the input file format of STeP (specifically to the `spec` file format).

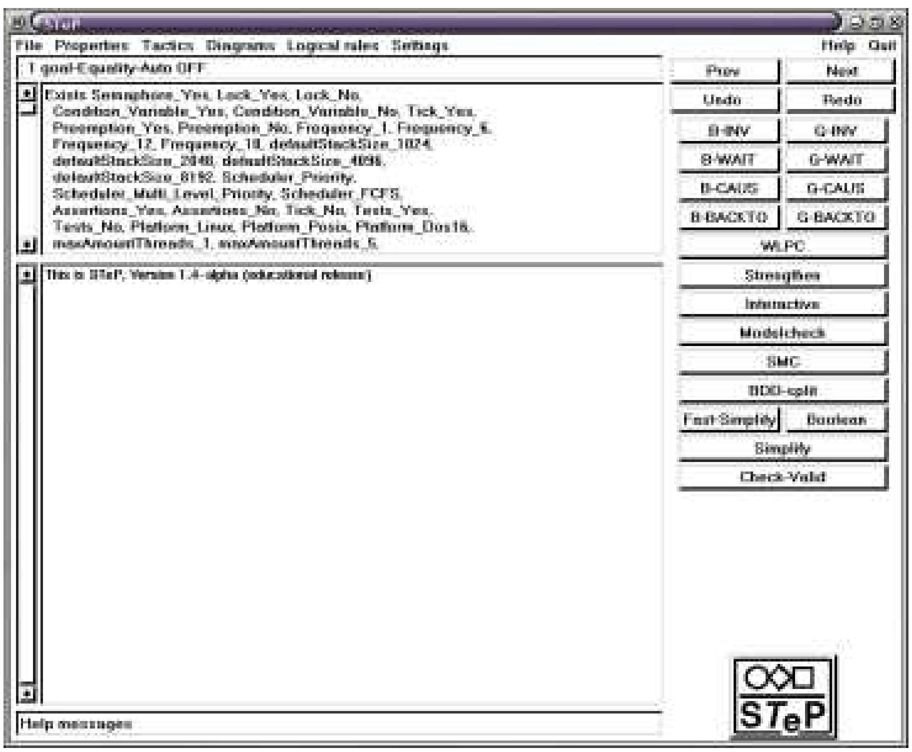


FIGURE 6.2.1. Screen-shot of STeP theorem prover with a restricted design space to check.

In the moment of this writing, we are using this theorem prover just to decide the existence of a model, we have not tried to use it to determine a model or to show a proof of the inconsistency. We think the enhancements of this tool include:

- A change of tool, since the current tool doesn't allow free redistribution and its sources are not distributed.

- In case of consistency find a model, to complete the profiling of a restricted design space if the user wants it.
- In case of inconsistency show a proof, and use it to highlight the correlations and dimensions that generate the inconsistency.

## 6.3. Integration with other tools

**6.3.1. Reboost.** Reboots is a tool developed by the groups B5 and B10 of the SFB501 project to maintain the information of a design space. Reboost is the predecessor of the tools presented in this work, and several of the key ideas of the restricted design space concept are already implemented in reboost (e.g. two boolean expressions for each correlation, types of correlations, hierarchies, groups).

The design spaces supported by Reboost are much more general than the restricted notion presented here, for that reason it is possible to translate any restricted design space to a design space supported by Reboost but not the contrary. In particular Reboost supports numeric ranges, a more general concept of group and sequence and some facilities to use groups with hierarchies. The only advantage of the restricted design space is the decidability of the consistency —precisely the objective of the restriction.

There exists an initial translator from Reboost databases to the XML file format (this translator ignores part of the extended design space that Reboost supports). Such translator is based on the work of Miko Lösch, and is written in Java and must be improved.

Regarding the implementation of translators, in addition to the theoretical obstacle mentioned, the biggest implementation obstacle is the differences in the platforms of both tools. While Reboost is written in Java with a database in Access for the Windows OS, the translators presented here were developed in Ocaml by using XML on the GNU/Linux OS. Although in theory all the tools are portable, the practice has been far from that.

We think that in this area, there are several enhancements to do:

- Ensure the portability of Reboost to run it at least on Windows and GNU/Linux.
- Ensure the portability of the translators and tools presented in this work to ensure they run at least on Windows and GNU/Linux.
- Complete the translator from a subset of the supported concept of Reboost to the XML file format.
- Implement a translator from the XML file format to the database used by Reboost.

**6.3.2. CML2.** CML2 is a language developed in the Open Source community to configure the kernel Linux; along with the language some tools were also developed to check the configuration rules and to perform a configuration[**19**]. The approach of CML2 is similar to the approach taken here: it also allows to declare constrains in the selectable options, and it also uses a decision procedure to check consistency.

To achieve results in reasonable time, the decision procedure of CML2 is not general; in accordance to the author it exploits facts about the domain of the Linux Kernel.

We have implemented a translator from the XML file format to the CML2 file format, hence it is possible to use the decision procedure of CML2 and the developed tools for that language. However, this tool has resulted inappropriate for our purposes since it cannot handle the sentences that define dimensions. For example the current decision procedure of CML2 cannot determine the insatisfability of the restricted design space presented in Figure 6.3.1.

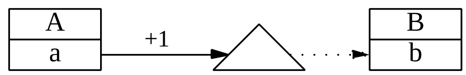


FIGURE 6.3.1. Example of a restricted design space not handled by CML2.

The translation of that design space to CML2 results in the following set of rules that erroneously is found consistent by the CML2 tools used (version 0.8).

```
# Options
symbols A_a 'a'
B_b 'b'
#Dimensions
menus A 'A' B 'B' firstMenu ''
choices A A_a default A_a
choices B B_b default B_b
#Hierarchies
#Correlations
require (A_a==y) implies (B_b==n)
start firstMenu
menu firstMenu A B
```

In this direction we think that some tasks to do are:

- There are rules to configure a Linux kernel (the version 2.4) written in CML2, we could try a translation of these rules or part of them to restricted design spaces, to test the concept with a bigger case.
- As well as Reboost, CML2 allows to use integer dimension and to set restriction on some calculated values, those are notions that could be useful and that should be formalized.

CHAPTER 7

# A complete example

We think, that the mapping from a restricted design space to source code, depends highly on the programming language, its configuration facilities and the style of the program to customize. In this area the XML file format facilitates the translation of a profiled restricted design space (or of a partially profiled restricted design space) to the methods supported by the target sources.

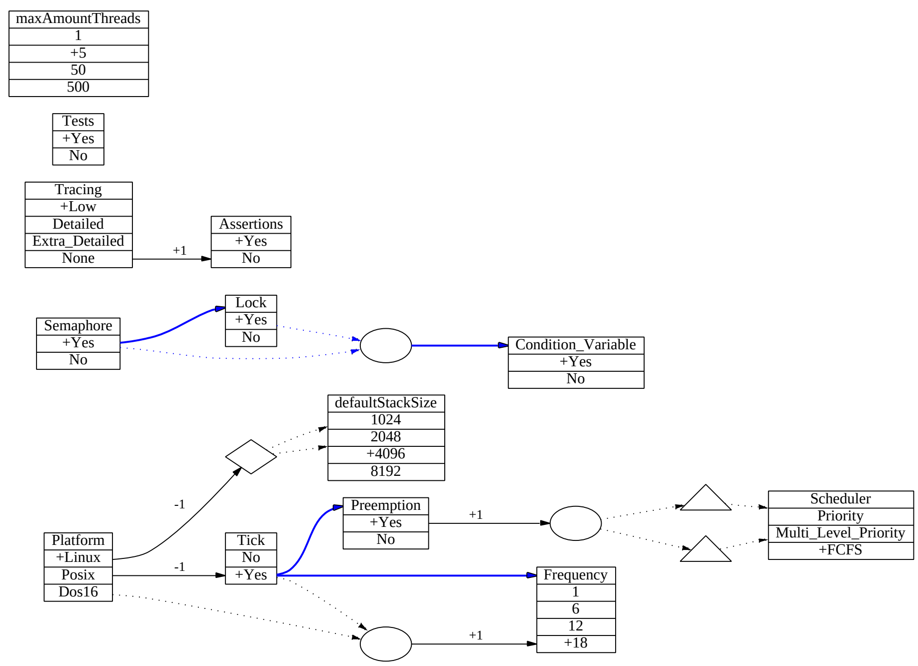


FIGURE 7.0.1. Design Space of GCRTK.

As an example the configuration of the kernel GCRTK has been done by translating its restricted design space (Figure 7.0.1) to a shell script that instantiates the sources.

GCRTK means Generic Component Real Time Kernel. It is a small kernel for embedded systems built with configurable components. The initial work, called RTK, presented in [**14**], was written in Assembler and C++. That author of this work re-implemented several parts of that version and implemented as mechanisms of variability: Makefiles, inheritance and the preprocessor (the initial sources only used the preprocessor to control the variability).

To configure the sources based on the selections in the design space, some variables of a shell script are changed (the technique is remotely inspired in Autoconf [**18**]). The resulting shell script is a generator/instantiator that applies several methods to configure the sources based on the value of the variables. The methods that the script implements are:

- Copying only selected parts of the hierarchy of directories and files to the target location.
- Generation of `Makefiles` and changing others (by using `sed`).
- Generation of small source files (for example the file `static.hpp`, that contains constants).
- After completing the instantiation, generation of the following documentation from the configured sources:
  - Inclusion graph.
  - Reference API, with description of every function in the configured sources.
- If tests were selected during profiling, and if the target platform is GNU/Linux: running regression tests tailored for the instantiated sources.

What follows is the graph of inclusions for two different selections of the restricted design space for GCRTK. Each node of these graphs is a source file, and each arrow indicates that certain file is included by other.

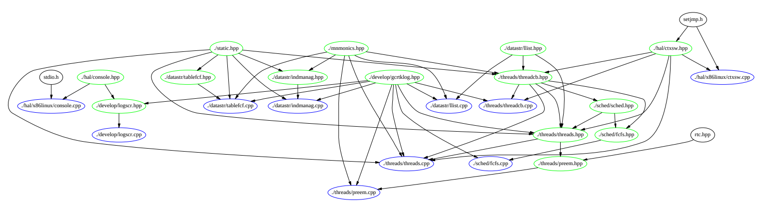

FIGURE 7.0.2. Graph of inclusions of a configuration of a minimal GCRTK.

The figure 7.0.2 shows the inclusion graph of a minimal GCRTK that runs emulated on GNU/Linux, that has no support for tracing, no support for assertions, it supports at most 5 threads, it doesn't have real time interrupt, it doesn't support preemption, the default stack size is 4096 and a "first come first served" scheduler.

The second graph (figure 7.0.3) presents a much more complex kernel. It was also configure to run on GNU/Linux, to support 50 threads, with semaphores, locks and monitors, with a real time clock, default stack size 4095 and a preemptive "first come first served" scheduler.

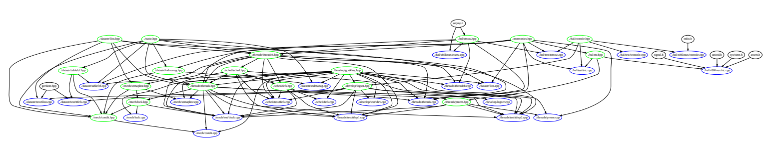

FIGURE 7.0.3. Graph of inclusions of a configuration of a complex GCRTK.

CHAPTER 8

# Conclusions

- A precise notion of restricted design space was formalized, and the equivalence of this definition with propositional logic was proved.
- Along with the formal definition we introduced a definition of consistency equivalent to satisfiability in propositional logic (hence equivalent to consistency in propositional logic). With the definition the consistency of a restricted design space can be checked in any stage of its development.
- Checking the consistency of a restricted design space is a decidable but NP complete problem. We have developed support to do it, for small restricted design spaces by using an automatic theorem prover.
- A graphical notation for restricted design spaces was given and tools to support it have been developed and used.
- A concrete syntax for a restricted design space was developed in XML.
- A restricted design space for GCRTK, a toy kernel for embedded systems, could be developed as first validation of the concept. We also developed a method to configure the sources of GCRTK based on the selections in its restricted design space, and a shell script as generator that adapts and modify the source code in several ways.
- There are several improvements to do in the developed tools, but probably the most important work to do is theoretical:
  - Restrict more the definition and find a decision procedure for the consistency that runs in polynomial time.
  - In case of inconsistency find a method to determine the parts of the restricted design space that have the problems.
- There are other areas to explore based on the formal definition given in this work:
  - Check the equivalence between different restricted design spaces.
  - Correlations and hierarchies are assertions that should be followed by the code associated with a design space. They could be used to test the conformance of a program in accordance with its restricted design space, or to add assertions automatically to a program or to generate test cases automatically.
  - Transform one representation of a restricted design spaces into a different but equivalent one (probably simpler).

# Bibliography


[1] Jon Barwise. *Handbook of Mathematical Logic*. North-Holland Publishing Company, 1978.

[2] Lothar Baum. Toward generating customized run-time platforms from generic components. In *Caise'99 6th Doctoral Consortium*, January 1999.

[3] Lothar Baum, Martin Becker, Lars Geyer, Andreas Gilbert, Georg Molter, and Vladimir Támara. Supporting component-based software development using domain knowledge. In *SCI 2000*, Florida, USA, 2000.

[4] Lothar Baum, Martin Becker, Lars Geyer, and Georg Molter. The role of architecture for complex systems development. In *Proc. of 11th International Conference on Software Engineering and its Applications*, Paris, France, 1998.

[5] Lothar Baum, Martin Becker, Lars Geyer, and Georg Molter. Using software architecture as a catalyst for reuse. In *European Reuse Workshop 98*, 1998.

[6] Lothar Baum, Martin Becker, Lars Geyer, and Georg Molter. A process view on architecture-based software development. In *Working IFIP Conference on Software Architecture*, Texas, February 1999.

[7] Lothar Baum, Martin Becker, Lars Geyer, and Georg Molter. Mapping requirements to reusable components using design spaces. In *4th International Conference on Requirements Engineering (ICRE-2000)*, 2000.

[8] Lothar Baum, Martin Becker, Lars Geyer, Georg Molter, and Peter Sturm. Driving the composition of runtime platforms by architectural knowledge. In *Proc. of 1998 SIGOPS European Workshop*, Sintra, Portugal, 1998.

[9] Lothar Baum, Lars Geyer, Georg Molter, Steffan Rothkugel, and Peter Sturm. Architecture-centric software development based on extended desing. In *Second Internation Workshop on Development and Evolution of Software Architectures for Product Families*, 1997.

[10] Lothar Baum, Lars Geyer, Georg Molter, Steffen Rothkugel, and Pter Sturm. Software develpment with "prêt-à-porter" systems. In *Symposium Object Oriented application Frameworks. SOOAF' 97*, 1997.

[11] Nikolaj Bjorner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, Marc Pichora, Henry B. Sipma, and Tomás E. Uribe. *The Standford Temporal Prover. User's Manual of version 1.4-a*. Standford University, http://www.research.att.com/sw/tools/graphviz/, 1998.

[12] Vijay Chandru and John Hooker. *Optimization Methods for Logical Inference*. John Wiley & Sons Inc., 1999.

[13] C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland Publishing Company, 1977.

[14] Marcel Ianizzi. Generisch betriebsystemimplementierung am beispiel eines realzeit-schedulers. Master's thesis, Universität Kaiserslatern, 1998.

[15] Eleftherios Koutsofios. *Editing Pictures with lefty*. http://www.research.att.com/sw/tools/graphviz/, 1996.

[16] Eleftherios Koutsofios and Stephen C. North. *Editing graphs with dotty*. http://www.research.att.com/sw/tools/graphviz/, 1996.

[17] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.00: Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, http://caml.inria.fr/ocaml/htmlman/index.html, 2000.

[18] David MacKenzie and Ben Elliston. *Autoconf: Creating Automatic Configuration Scripts*. http://www.gnu.org/manual/autoconf, 1998.

[19] Eric Steven Raymond. *The CML2 Language: Constraint-based configuration for the Linux kernel and elsewhere*. http://www.tuxedo.org/ esr/cml2/cml2/cml2.html, 2000.

[20] Gerd Stolpmann. *The PXP user's guide.* http://www.ocaml-programming.de/packages/documentation/pxp/manual, 2000.

[21] Lane T.G. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, Carnegie Mellon University, 1990.

# Appendix A

```
<!-- DTD for XML file format for restricted design spaces -->

<!ELEMENT dsx (#PCDATA | dimension | correlation | hierarchy | sequence)*>
<!ATTLIST dsx
           ver0_3 CDATA #REQUIRED>
<!ELEMENT dimension (#PCDATA|option)*>
<!ATTLIST dimension
          id CDATA #REQUIRED
          id_group CDATA #IMPLIED>
<!ELEMENT option (#PCDATA)>
<!ATTLIST option
          id CDATA #REQUIRED
          selected CDATA #IMPLIED>
<!ELEMENT correlation (expr,expr)>

<!ATTLIST correlation
          symmetric CDATA #REQUIRED
          positive  CDATA #REQUIRED
          description CDATA #IMPLIED>
<!ELEMENT hierarchy (expr)>
<!ATTLIST hierarchy
          description CDATA #IMPLIED
          idDimension CDATA #REQUIRED>
<!ELEMENT expr ( and | or | not)?>
<!ATTLIST expr
          id_opt CDATA #IMPLIED>

<!ELEMENT and (expr,expr)>

<!ELEMENT or (expr,expr)>

<!ELEMENT not (expr)>

<!ELEMENT seq ( seq)*>
<!ATTLIST seq
          id_dim CDATA #IMPLIED>
```