Multidimensional Tree-Structured Spaces for Separation of Concerns in Software Development Environments

Doug Kimelman

IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights NY 10598, USA
dnk@watson.ibm.com

Preface

This paper proposes multidimensional spaces as an organizational paradigm for purposes of multidimensional separation of concerns ("MDSC"). The ideas were originally conceived of for organizing modules, and other artifacts more generally, within environments for software development and other related pursuits. Nonetheless, we believe that the ideas are entirely applicable to the more specific case of organizing the methods of large, evolving, production object-oriented software systems. Thus, this paper is presented as an argument by analogy, in support of MDSC for object-oriented systems.

This work builds on familiar notions, such as multidimensional spaces and projections, from analytic geometry and linear algebra, in order to achieve a formalism that is powerful yet approachable to practitioners — users who, while unfamiliar with specific notions and terminology from emerging fields such as MDSC, nonetheless might well be familiar with basic mathematical concepts.

The paper thus constitutes a contribution towards a theoretical foundation for the organizational aspects of multidimensional separation of concerns. It includes discussions related to: the need for multidimensional separation of concerns, problems arising out of inadequate separation of concerns, multidimensional separation of concerns throughout the software lifecycle, and concerns that span lifecycle phases and software artifacts.

Abstract

Current environments are inadequate for organizing the modules and other artifacts of large scale development projects.

This paper proposes "multidimensional tree-structured spaces" ("md-nt" spaces) as a powerful and general means of organization within development project environments.

The multidimensional nature of md-nt spaces allows the formalization of the distinct and orthogonal nature of such structures as: hierarchies of modules, networks of revisions, and collections of variations of modules. Extensions to the basic md-nt structure allow it to accommodate the complexities and irregularities often encountered in actual practice.

Semantics, such as implications in terms of module nesting or revision sequence, can be dynamically associated with an md-nt space. This enables many common system functions, such as recompilation of source, or storage and regeneration of revisions, to occur automatically.

Introduction

Computer systems are the primary facilities underlying development projects in a number of different areas. Such areas include software development, VLSI design, web site creation, document preparation, and graphics generation.

Projects in each of these areas are often based on large collections of information. This information constitutes the "materials" for the development process. Such information is typically maintained in a set of files, in a database, or in a structure internal to a development environment.

Most projects exhibit a number of common requirements concerning the organization of this information. These requirements include the need to accommodate: hierarchies of modules (in the most general sense), networks of evolutionary revisions of modules, collections of alternative variations of modules, and series of successive forms or intermediate representations through which a module progresses.

This paper proposes “multidimensional tree-structured spaces” (“md-nt spaces”) as a powerful and general means of organizing development project file systems.[1]

A file system which incorporates such a structure is regarded as a multidimensional space, and each file of the system is regarded as a point in the space. Each axis of the space is a tree, or a singly-rooted network. Thus, each axis provides an alternative, or complementary, hierarchical organization for the points of the space.[2]

The multidimensional nature of an md-nt space allows the formalization of the distinct and orthogonal nature of considerations such as modular decomposition, revision, variation, and successive transformation. Extensions to the basic structure of an md-nt space allow it to accommodate the file system complexities and irregularities that are often encountered in actual practice.

The following sections discuss in greater detail: development project requirements concerning file system organization; the deficiencies of contemporary systems in satisfying these requirements; the structure of md-nt spaces; high-level operations on md-nt spaces; and declaration of the semantics of md-nt spaces.

Requirements Concerning Organization

There are a number of requirements concerning file system organization, which are common to most large scale development projects.

- Such projects often require that information be organized into elaborate hierarchies of modules. Figure 1 illustrates such a hierarchy.
- There is often a series of successive forms, or intermediate representations, through which a module progresses. Figure 2 illustrates a hierarchy of such forms. These forms result from the various analyses, translations, and other automated transformations that may be applied to a module. Other forms of a module are associated with other stages in the development process. These often result from manual processes or less formal derivations. Each of the forms in which a module may exist is referred to here as a "phase" of the module. Note that, in many cases, a collection of such phases is inherently hierarchical in nature.
- For most projects, there are typically numerous evolutionary versions of the various modules. These versions are commonly referred to as "revisions".

  For a given software module, there might be a main stream of revisions, as well as a number of divergent branches of revisions. The main stream would result from successive fixes and minor improvements that occur as a natural part of the evolution of the module. The branches might reflect experimentation or more significant development efforts due to major enhancements. Some of a module's revision branches are eventually abandoned, while others are eventually merged back into the main stream, or into some other branch. Thus, the revisions of a particular module often form an elaborate network, rather than a simple sequence or a conventional hierarchy. Figure 3 illustrates such a network.
- As well as evolutionary versions of a module, there are often alternative versions. These versions are commonly referred to as "variations".

  A variation of a software module might exist for each machine which will eventually act as host to the resulting code. Figure 4 illustrates a hierarchy of such variations. Variations might also exist according to the set of selectable features that have been chosen for a particular incarnation of a software system. Figure 5 illustrates a hierarchy of such variations.

In general, it can be seen that development projects have relatively demanding requirements concerning the organization of the information on which they are based. It seems that a fundamental requirement is to organize files according to each of a number of distinct considerations, hierarchically in each case, with one consideration independent of the other.

[1] Although md-nt spaces are presented here within the context of file systems, they could just as well serve as the fundamental structure underlying persistent object repositories for object-oriented systems, or collections of abstract syntax trees for language-directed environments.

[2] In fact, md-nt spaces are based on general networks rather than strict hierarchies. However, it is expected that in actual practice, trees, rather than full networks, will predominate. Therefore, general discussions within this paper consider the case of hierarchies. More specific discussions, however, such as those concerning the organization of revisions and releases *do* consider the case of networks.

Organizational Deficiencies of Contemporary Systems

Current file system structures are inadequate for organizing development project information.

Tree-structured file systems (as in Unix / Linux) attempt to represent many different organizational properties of a collection of files with a single hierarchical structure. Pathnames such as '/ u / smith / mydbms / cmds / source / 604e / newversion / util_fltpt.c' are not uncommon. Ad-hoc conventions concerning file name suffixes are often employed to convey additional information about file attributes. With such "overloading" of the primary file system structure, the file hierarchy becomes extremely cluttered, the manipulation of files becomes cumbersome, and the maintenance of large programs becomes complex and error-prone.

Utilities such as CVS or RCS are often used to maintain all of the versions of a particular source file in a single space-efficient archive (incorporating a differential base + delta storage scheme). Such utilities often employ custom organizations and command languages which are foreign to the standard environment. These organizations and languages tend to be ad-hoc, limited, and restrictive. The fact that the versions must be accessed by special-purpose commands, and cannot be manipulated by the standard utilities, further complicates maintenance procedures.

Facilities such as Make require dependency information in special dependency files or configuration specifications. Such dependency information is external to the primary file system structure. Thus, it is potentially redundant, and invites inconsistencies. As well, the standard system utilities are often not aware of such information. Thus, this information is unavailable, or fails to be applied automatically, in many situations where it might be of further use. Such isolation of information is contrary to the goal of an integrated development environment.

In general, conventional systems such as these fail to take full advantage of whatever information can be inferred from the file system organization that does exist. Users of such systems must interpret the file system structure manually, in order to explicitly direct the manipulation and processing of the various files. Thus, procedures which could be formalized and automated are accomplished manually in an ad-hoc and error-prone fashion.

Many such systems exhibit the following deficiencies:

- They do not formalize specific organizational considerations. For example, they would make no formal distinction between variations due to host machine and variations due to algorithm.
- Classifications with respect to a particular consideration are essentially unordered. For example, the set of classifications 'PowerPC', '405', '604e', '750', 'Pentium', 'PIII', and 'PII' would occupy a single level of a tree, immediately below a node of variation corresponding to the consideration 'host'. Hierarchies of classifications with respect to a particular consideration are not directly supported.
- Classifications are not specified with respect to, or qualified by, the consideration to which they apply. For example, nodes may be selected based on a node identifier such as 'convergent', regardless of whether the identifier refers to a host machine, an algorithm, or a revision.
- Structures reflecting certain considerations may have to be replicated in order to accommodate the various possible combinations of classifications. For example, the host machine hierarchy may have to be replicated under each of a number of nodes in the module hierarchy, or vice versa.

Md-nt Spaces

"Md-nt spaces" constitute a powerful, general, and formal means of organizing the files of large scale development projects. They resolve many of the deficiencies identified in the previous section, and satisfy the organizational requirements discussed earlier.

A development project file system which is organized as an md-nt space is regarded as a multidimensional space, and each file of the system is regarded as a point in the space. Thus, the file system may be regarded as a "space of files ", or a "file space".

The remainder of this section considers the more abstract case of "points" within "spaces". These discussions, however, apply identically to any concrete form of object within the corresponding form of space (one example being that of organizing all of the methods of a large, evolving, production object-oriented software system, along with the all of the other artifacts of the development process, all within an encompassing space). Further, any such space can be realized equally well using an extended file system, an extended relational database, or a suitably structured repository within an object-oriented development environment.

Fundamentals

Figure 6 depicts a collection of points organized within the framework of a conventional 3-dimensional space. The space has three dimensions, each dimension has an axis, and each point has three coordinates. The space is "conventional" in the sense that its axes are linear, rather than hierarchically structured.

The space's three dimensions are: 'module', 'phase', and 'host'. The axis for the module dimension, or the "module axis", is depicted by the solid line marked with the values: 'compiler', 'scanner', 'parser', and 'symmgr'. The point labeled by [parser ; obj ; 750] has 'parser' as the value of its module coordinate, the value 'obj' for its phase coordinate, and the value '750' for its host coordinate.

It is intended here that each dimension reflect, or correspond to, a distinct organizational consideration. For example, for the space depicted in Figure 6, the 'module' dimension organizes points according to the module to which they correspond. The 'phase' dimension organizes points according to the phase to which they correspond, and the 'host' dimension organizes points according to the host machine to which they correspond. Naturally, for a development project file space, the number of dimensions, and the considerations to which they correspond, should be dynamically definable by the user.

Further, each axis of a space should constitute the domain of possible classifications for a point, with respect to the consideration of the axis' dimension. For example, for the space of Figure 6, the possible classifications for a point with respect to the module dimension, or the possible modules to which a point may correspond, are: 'compiler', 'scanner', 'parser', and 'symmgr'.

Finally, the value of a point's coordinate in a given dimension should constitute that point's classification with respect to the given dimension's consideration. For example, in Figure 6, the point labeled by [parser ; obj ; 750] is classified as, or corresponds to, the object code phase, of the parser module, for a PowerPC 750.

Axis Structure and Point Ordering

The values along an axis of a space can be ordered. For example, for the space of Figure 6, the values of each axis are linearly ordered. Thus, as well as constituting a set of possible values for coordinates with respect to a given dimension, an axis also provides an ordering for these values.

Further, the points of a space can be regarded as being ordered according to their coordinate values. This ordering is based on the ordering of values established by the axes of the space. For example, for the space of Figure 6, the point [parser ; obj ; 750] could be regarded as preceding the point [symmgr ; obj ; 750] with respect to the module dimension, because the value 'parser' preceded the value symmgr on the module axis.

Thus, the points of the space are not just *classified* with respect to each of a number of considerations, they are also *ordered* with respect to each of those considerations.

Note that the organization of points with respect to one consideration is entirely independent of the organization of those points with respect to any of the other considerations. More formally, the axes of the space are orthogonal, and the space itself is referred to as an orthogonal space.

Also note that the space of Figure 6 has linear axes. Thus, although it does organize points with respect to each of a number of distinct considerations, with each consideration being independent of the other, it does so linearly. The fundamental organizational requirement identified earlier was to organize these points hierarchically. Hence, tree-structured axes are introduced in the next subsection.

Note, however, that the general structure on which md-nt spaces are based is in fact a network rather than a strict hierarchy. The structure is a directed graph, with exactly one distinguished "root" node having indegree 0. The constraint to a single root is imposed for purposes of unambiguous qualification of node identifiers, and for purposes of display of these structures. It is expected however, that the case of strict hierarchies, rather than full networks, will prevail in actual practice.

Tree-structured Spaces

Figure 7 depicts a 2-dimensional tree-structured space. The space has two dimensions, each dimension has an axis, and each point has two coordinates. The space is "tree-structured" in the sense that its axes are tree-structured rather than simply linear. The space's two dimensions are: 'module' and 'phase'. The axis for the module dimension is the tree depicted by the horizontally oriented solid lines, marked with such values as 'compiler', 'syn', 'bid', 'getc', and 'act'.

Each axis of the space now provides a hierarchical ordering for its values. Thus, each axis now imposes a hierarchical ordering on the points of the space. As well, the space now organizes points, according to each of a number of distinct considerations, *hierarchically* in each case, with each consideration independent of the other.

Note that each value along an axis is now fully identified using a pathname rather than a simple identifier. For example, for the space of Figure 7, the axis value labeled 'act' can be referred to using the fully qualified pathname 'compiler . lex . scan . act'[3].

Given that an axis constitutes a domain of possible values for point coordinates, each coordinate of a point now has a value that is identified by a pathname. Further, each point is now uniquely identified by a pathname m-tuple. For example, for the space of Figure 7, the point labeled [module: compiler . lex . scan . act ; phase: pgm .obj] has the coordinate value 'compiler ... act' in the module dimension, and the coordinate value 'pgm . obj' in the phase dimension.

Perspectives

Note that, in one sense, the space of Figure 7 could be regarded as consisting of a collection of phase trees, with cross-links joining points which have identical module coordinate values. In another sense, the space of Figure 7 could be regarded as consisting of a collection of module trees, with cross-links joining points having identical phase coordinate values.

Also note that, from "below", the space of Figure 7 might appear to be a single module tree, with a phase tree growing straight out "behind" each module. Conversely, from the "left side", the space might appear to be a single phase tree, with a module tree growing out of each phase.

These alternative perspectives regarding the structure of an md-nt space are an important part of the users' overall model of the organization of their files. These perspectives contribute greatly to the ease and power with which a user can manipulate the space and deal with its points.

Operations on Md-nt Spaces

High-level operations on md-nt spaces have been defined, in order to allow a user to deal with the global structure of a space.

Projection

The "projection" operation allows a user to deal with points, in groups according to their coordinate values in a given set of dimensions, without regard for their coordinate values in other dimensions. For example, a single module tree results from projecting the space of Figure 7 onto its module dimension. Each node of this module tree may be regarded as having a tree of points, identical in structure to the phase axis, growing straight out "behind" it. When users deal with a node of this module tree, they are actually dealing with the tree of points corresponding to the entire phase tree for the given module. Thus, for example, moving 'getc' of this module tree to a new node under 'scan', actually moves all of the phases for 'getc', including its source, object code, and "Make" instructions, at once. That is, points are being dealt with in groups according to their module coordinate values, without regard for their phase coordinate values.

Conversely, projecting the space of Figure 7 onto its phase dimension would result in a single phase tree. identical in structure to the phase axis. Applying the 'delete contents' operation to the node 'obj' of that phase tree would delete the object code for all modules at once.

Flattening

The "flattening" operation allows a user to arrange all of a space's points into a single hierarchy (one of a number of possible single hierarchies). The single hierarchy is derived according to a user-specified ordering of the space's dimensions. For example, nesting phase within module results in a hierarchy formed by taking the module hierarchy and replicating a phase hierarchy to be placed under each module. A user will employ a particular

[3] Naturally, for a development project environment, users should be allowed to indicate a "current context", and then to use partially qualified pathnames that are unambiguous with respect to that current context.

flattening if it suits the development task at hand particularly well. Flattening also facilitates convenient display of the global structure of md-nt spaces using textual views.

Slicing

The "slicing" operation allows a user to isolate a contiguous region of a space. For example, a layer of the space results from slicing the space of Figure 7 across the module dimension, and "along" the axis values bounded by 'src', 'defn', and 'impln' of the phase dimension. Projecting that slice onto its module dimension results in a single module tree. Note that *projecting* the entire space onto its module dimension also yielded a single module tree. Now, however, each node of the module tree represents only those points corresponding to the source subtree of the phase tree for the given module, rather than the points corresponding to the entire phase tree. Thus, applying the 'string search' or 'print' operation to the nodes 'lex', 'scan', and 'getc', would search or print only the points corresponding to the source phases of those modules.

Note that whereas projection and flattening provide alternative views of an entire space, slicing isolates a subset of a space's points.

Also note that although a user might traverse a space in only one dimension at a time, or might deal with only a few dimensions at a time, or might choose to view a space as a single hierarchy, the user is free at any instant to choose an entirely different orientation. That is, the user is always free to choose a different dimension in which to move, or a different set of dimensions to deal with, or a different nesting of dimensions with which to form a hierarchical view.

Thus, the power of the md-nt structure, in terms of operations on the space, lies in the generality and the flexibility of the viewing and manipulation that it supports. The high-level operations allow the suppression of burdensome structural detail, and the formulation of views that are tailored to the development task at hand. When performed in the context of a suitable view, tasks which are quite complicated with conventional systems, become straightforward.

Extensions to the Basic Structure

Md-nt spaces, as discussed above, are strictly orthogonal. That is, the organization of points with respect to one consideration, or in one dimension, is entirely independent of the organization of those points with respect to any of the other considerations or dimensions. More formally: a point's coordinate in one dimension is entirely independent of its coordinate in any other dimension.

However, there are file system complexities that arise in actual practice, that cannot be accommodated smoothly by a simple orthogonal structure. Hence, extensions to the basic md-nt structure are introduced.

For example, for a large software system, different modules often have different evolutionary paths. In this case, the notion of a single revision hierarchy that is common to all modules, is inappropriate. Instead, a distinct organization of revisions is required for each module.

This is formalized by the notion of a "varying axis structure". That is, each partition of a space, defined by particular coordinate values in a "determining" dimension, is allowed to have a distinct axis structure in a "dependent" dimension. Each such distinct structure is referred to as an "axis variant".

As well, although the axis of a given dimension may vary according to partition, there may still be a need for a common general organization of the space's points with respect to that dimension. For the space discussed above, there might be a requirement for a common general view of the various revision organizations. For example, it may be necessary to view the revisions of each module in terms of the system release to which they correspond.

In this case, a hierarchy (or network) of releases could be formed, which could be regarded as the most general view of any module's revision hierarchy. This network could be regarded as an "overlay", which could be mapped to any revision axis variant, but which would "fit" slightly differently onto each variant.

Semantics of Md-nt Spaces

Significant benefit could result from informing the system about the meaning that users associate with the various aspects of the structure of an md-nt space.

For example, a user could be allowed to declare that links in a given dimension interconnect modules which are in fact revisions of one another. This would allow the system to automatically store only the difference between a parent, such as [syn ; 1], and a child, such as [syn ; 1.1]. Further, depending on the specific semantics desired for a particular space, any changes made to an older version of a file could automatically be propagated to its successors.

Another example of automatic consequences of declared semantics is recompilation and relinking whenever source is modified or an outdated executable is fetched. Such actions could result from dependency semantics declared for dimensions such as ‘module’ and ‘phase’.

Summary

This paper has proposed the hierarchical classification of objects according to each of a number of distinct considerations, as a paradigm for organization within a development project environment. This is formalized by the definition of a multidimensional hierarchical structure.

The md-nt structure allows the formalization of the distinct, orthogonal, and hierarchical nature of such structures as hierarchies of modules, networks of evolutionary revisions, collections of alternative variations, and series of successive forms or representations through which a module progresses.

High-level operations on md-nt spaces enable the suppression of burdensome structural detail. Further, such operations allow the formulation of arbitrary views of the file space. When performed in the context of a suitable view, numerous tasks which are quite complicated with conventional file systems, become straightforward.

The declaration of semantics enables many common system functions to occur automatically based on the structure of the space. In the past, such operations have been accomplished in a manual, ad-hoc, and error-prone fashion, or via mechanisms which were external to the primary environment structure, and hence contrary to the goal of an integrated environment.

Thus, md-nt spaces constitute a highly effective means of organizing the modules (in the most general sense) and other artifacts of large scale development projects.

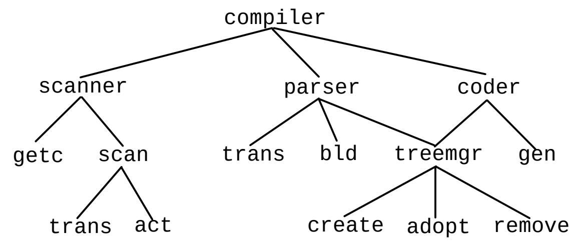

Figure 1: A module hierarchy

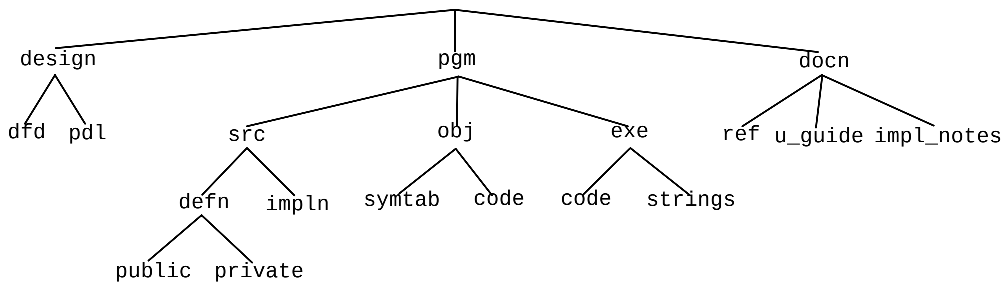

Figure 2: A phase hierarchy

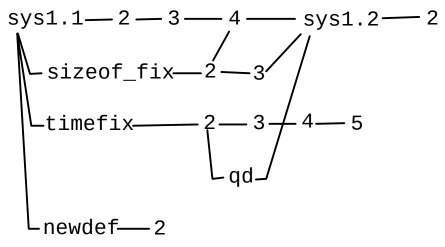

Figure 3: A revision network

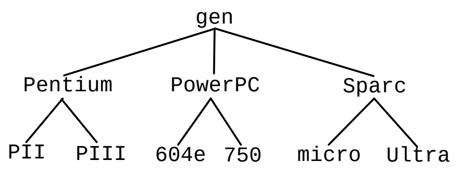

Figure 4: A host hierarchy

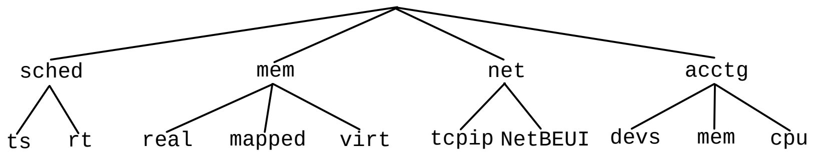

Figure 5: A feature hierarchy

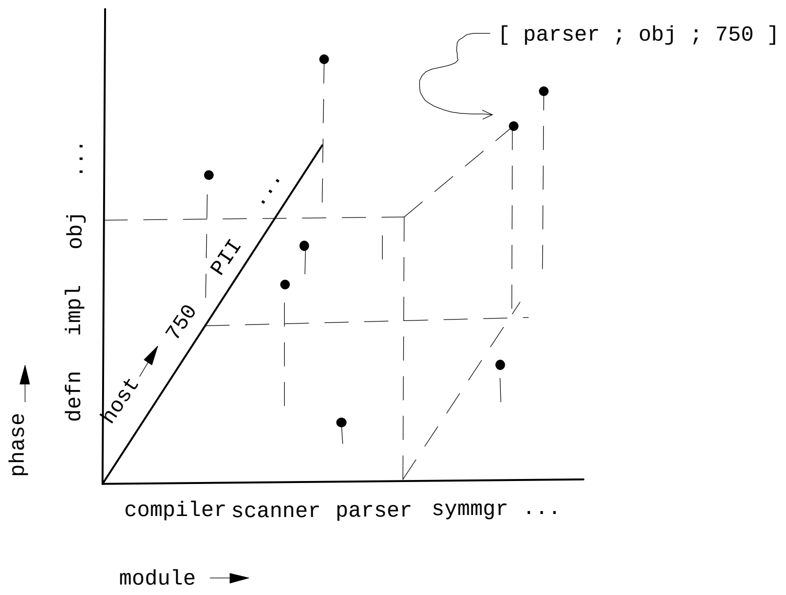

Figure 6: A 3-dimensional space (module x phase x host)

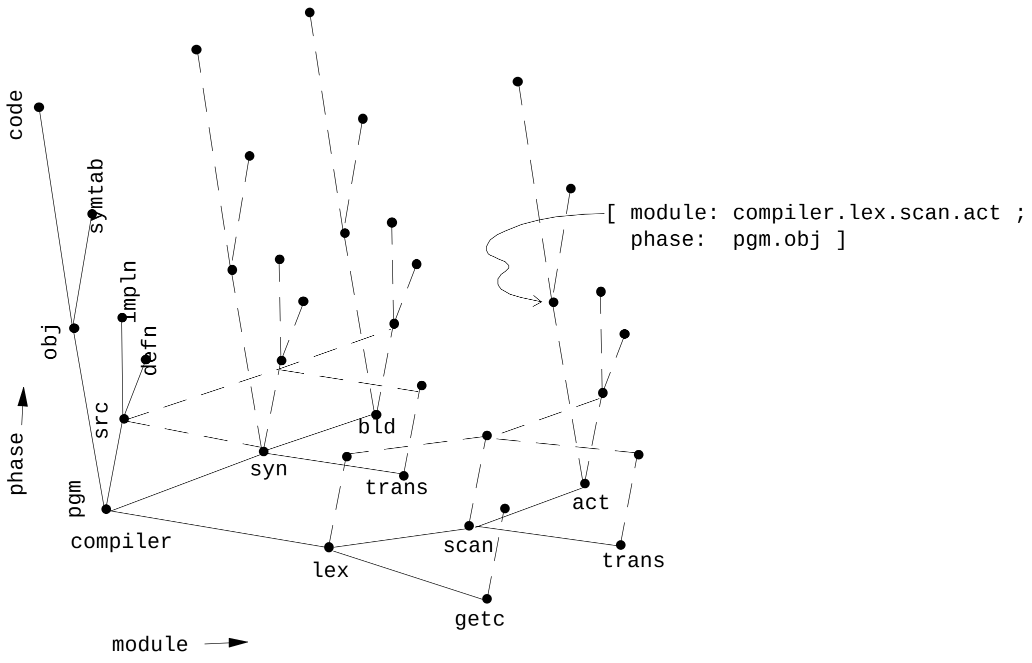

Figure 7: A 2-dimensional tree-structured space (module x phase)