Language Parametric Module Management for IDEs

P. Klint [1] A.T. Kooiker [2] J.J. Vinju [3]

Centrum voor Wiskunde en Informatica
P.O. Box 94079, NL-1090 GB
Amsterdam, The Netherlands

Abstract

An integrated development environment (IDE) monitors all the changes that a user makes to source code modules and responds accordingly by flagging errors, by re-parsing, by rechecking, or by recompiling modules and by adjusting visualizations or other information derived from a module. A *module manager* is the central component of the IDE that is responsible for this behavior. Although the overall functionality of a module manager in a given IDE is fixed, its actual behavior strongly depends on the programming languages it has to support. What is a module? How do modules depend on each other? What is the effect of a change to a module?

We propose a concise design for a language parametric module manager: a module manager that is parameterized with the module behavior of a specific language. We describe the design of our module manager and discuss some of its properties. We also report on the application of the module manager in the construction of IDEs for the specification language ASF+SDF as well as for Java.

Our overall goal is the rapid development (generation) of IDEs for programming languages and domain specific languages. The module manager presented here represents a next step in the creation of such generic language workbenches.

1 Introduction

The long term goal of our research is generation of Integrated Development Environments (IDEs) for programming languages and domain specific languages. This is a classical topic, with a traditional focus on the generation of syntactic and semantic analysis tools. In this paper we focus rather on generating the *interactive* behavior of IDEs.

[1] Email: `P.Klint@cwi.nl`
[2] Email: `A.T.Kooiker@cwi.nl`
[3] Email: `J.J.Vinju@cwi.nl`

1.1 Motivation

IDEs increase productivity of programmers by providing them with an efficient input interface and rapid feedback. For many software projects the availability of a good IDE is one of the decisive factors in programming language selection. With language design and domain specific languages (DSLs) back on the (research) agenda [5], and knowing that tool support for DSLs is one of the limiting factors for their application [13], the key question is: "What is the quickest way to construct a full-fledged IDE for any kind of language?"

IDEs are complex systems. Apart from editing, building, linking and debugging programs they offer syntax highlighting, auto-completion, formatting, outlining, spell checking, indexing, refactoring, context-sensitive help, advanced static analysis, call graphs, version control, round-trip engineering, and much more. Programming languages have become more complex and software products are getting bigger and bigger. Many products actually use multiple programming and domain specific languages. This all adds up to the complexity of IDEs and building them requires major investments as exemplified by the effort in constructing Eclipse [6], and its various instantiations for Java, C, Cobol, and other languages.

The subject of this paper is a central part of each IDE that we call the "module manager". The module manager coordinates all actions within the IDE and all interaction with the programmer. It does this by responding to the changes that the programmer makes to the source code of a project, and by triggering "actions" accordingly. The module manager does not implement the actual interaction with the user, nor does it implement any specific action, but it does coordinate these actions. The main data model behind such coordination is the collection of source code modules of a software project and their interdependencies. A well-designed module manager is central to each IDE and reduces the coupling between other components. It leads to a plug-in architecture in which IDE components can be added independently.

The mother of all module managers is the tool *make* [8] that uses the module dependency graph to initiate build actions on source code modules in a batch-like fashion. The functionality of a module manager for an IDE is, however, much more complex. It has to react to many external triggers, is not restricted to pure build actions, and has to initiate many different actions as well. Examples are parsing, checking or compiling of modules, and adjusting visualizations or other information derived from modules such as context-sensitive help and error lists. The module manager is a fully interactive scheduler. It knows about language semantics in terms of modularity and packaging, and it knows about the capabilities of the IDE in terms of input and output to the user-interface. The main goal of the module manager is to provide fast and adequate feedback to the programmer on any modification she makes to any module's source code.

1.2 A Language Parametric Module Manager

The basic functionality of a module manager is to provide access to the modular structure of the source code of a software project. This "modular structure" is different for each language. Apart from their pure syntactic appearance, the meaning of modules and module dependencies differs per language. For instance, the include mechanism of the C preprocessor does not coincide with a C namespace; files are simply concatenated one after the other. The Java import mechanism, however, does coincide with the namespace of a compilation unit; a class can be made invisible outside the compilation unit it is defined in. Another example: Java has wildcards in import statements, a feature that is not present in C. The "module semantics" of a language is an important aspect of its syntax and semantics that is essential from the viewpoint of IDE construction. Large applications may even contain circular module dependencies: consider the processing of a text document containing an embedded spreadsheet that in its turn contains a text document, the syntax definition of a language in which statements can contain expressions but expressions may contain statements as well, or various design patterns that result in circular module dependencies.

Our goal is to develop a module manager that supports rapid prototyping of IDEs for any (domain specific) language and satisfiesthe following requirements:

R1 (*Language parametric*) It should be parameterized with the "module semantics" of a language. Circular module dependencies should be allowed.

R2 (*Schedule actions/rapid feedback*) It should schedule actions, optimizing the schedule for rapid feedback to the programmer.

R3 (*Open*) It should be open and be able to share a (partial) view of the modular structure of a project with other parts of the IDE.

R4 (*Scalable*): It should scale to large applications.

1.3 Contributions and Road Map

This paper contributes the following ideas:

- The use of *attributed module dependency graphs* as a practical and efficient vehicle for implementing a language parametric module manager.
- The use of *a simple modal logic* as a way to parameterize a module manager with language specific module semantics.
- An efficient algorithm for implementing this logic on top of an attributed module dependency graph.

In Section 2 we define the functionality of a model manager and its underlying data model. Section 3 gives an overview of the architecture of our implementation of such a module manager. In Section 4 we highlight the efficient implementation of the modal logic. Section 5 describes the case studies in which we applied our module manager to construct various IDEs. Section 6 summarizes our conclusions.

2 Attributed module dependency graphs

2.1 Basic representation

Directed graphs are an obvious representation for programming language modules and their interdependencies. We identify the vertices of a graph with the modules of a program, and the edges of the graph with the dependencies between the modules of a program. Each node has a unique name and a collection of attributes. Each attribute has a unique name within the scope of the node, and an arbitrary value. Dependencies are anonymous but they do have attributes that allow the distinction between different types of dependencies.

We call the modules that depend on module M the parents of M and we call the modules that module M depends on the children of M. Graphs can contain cycles and we can therefore represent cyclic dependencies.

Let's consider two examples. In Java, modules could be "classes", "packages" and "compilation units". Classes and packages are identified by their qualified name (i.e., including package prefix) and compilation units are identified by filename. Java has dependencies of type "containment", "import", and "inheritance". Classes are contained in compilation units or other classes, compilation units are contained in packages, and packages are contained in other packages. Classes import other classes, and inherit from other classes.

In C, modules could be "compilation units" and "header files", both are identified by filename. For dependency types C has "includes" and "uses external declaration". Compilation units and header files can include each other, and they can declare dependencies on anonymous compilation units via external declarations.

2.2 Mapping Language Concepts to the Graph Model

The mapping of programming language concepts to our graph model is rather arbitrary and depends on the granularity of interaction required by the IDE. For example, "functions" in C could be considered to be "modules" that depend on each other via a "calls" dependency. The only reason for labeling a programming language artifact as a module should be that the IDE needs the knowledge about dependencies between these modules to trigger certain actions.

2.3 Attributes

Modules and dependencies may have arbitrary attributes. For a specific programming language, there are specific attributes that will be used by the IDE to implement language specific behavior. Module attributes will be used to visualize a module's identity to the programmer. For example, in a Java IDE a class module will have a class name attribute and a package name attribute. Other attributes may contain aggregated information, such as whether a module contains a syntax error or not, or how many lines of code it spans.

2.4 Namespaces

One of the complexities of today's IDEs is that they have to deal with several programming languages and domain specific languages that are either operating next to each other or are embedded in each other. To be able to support several concepts of module semantics at the same time, we introduce *namespaces* for all identifiers in our graph based model. So, module identifiers, dependencies, module attributes, and dependency attributes all have a namespace. For brevity, we will assume from now on that a valid namespace is part of each module or attribute identifier.

2.5 Events

So far, we have only introduced a generic data structure for storing and retrieving transient information about modules. In order to schedule actions we need rules to select actions for execution. Examples of actions are "compiling" a compilation unit, or "extracting" an outline of a Java class, "alerting" the programmer about a certain error, or "decorating" a package view with versioning pictograms. The rules of the module manager should trigger these actions at the appropriate times.

The *listener design pattern* is a simple method for decoupling coordination from computation. A computation, or action, registers itself as a listener, and the coordinator triggers the action at certain moments. The module manager allows registration of listeners to *attribute change events*, *module existence events* and *dependency existence events* such that an action may be triggered on any change in the data model. Note that actions may influence the state of the module manager, triggering new actions. Since we do not assume anything about the actions, there can be no a priori guarantee that such a process would terminate, not deadlock, or even be deterministic.

2.6 Module Predicates

As we have seen earlier, "make" triggers build actions using the dependency relationship between modules. For example, the module graph contains the basic information for recompiling parts of a Java program without rebuilding the rest of it. In an IDE there are much more actions to be triggered under different kinds of conditions. For example, if a method is removed from a Java class, outlines need to be recomputed for all classes that inherit from it. Or, if a C include statement is moved in a file, at least all code between the old location and the new location needs to be rechecked for static correctness. Or, if a Java compilation unit is "modified" (in terms of the version management system), then all packages it is contained in are also "modified".

The information that needs to be propagated through a module dependency graph is language specific, even IDE specific. So, the module manager must provide some way of making this information propagation programmable. For this we introduce *module predicates*. These are inspired by attribute grammar systems [14] and modal logic [2]. Both formalisms provide a programmable way of distributing information over the elements of a complex data structure. An example of a module predicate for a C IDE is “linkable”. A C compilation unit is linkable when it contains a main function and all of its dependencies have compiled correctly.

We will get back to the details later in Section 4. For now, the key idea is that the truth values of module predicates are determined automatically by inspecting and aggregating the values of the attributes of a module and possibly other modules. The way this inspection and aggregation is done is determined by *module predicate definitions*, which the module manager receives at configuration time. When the value of such a predicate is changed as a result of the changed value of some attribute, a *predicate changed event* triggers the appropriate actions via listeners. The definitions are expressed using a simple logic, which allows the module manager to statically check for consistency of the set of definitions.

2.7 API of the Module Manager

The basic operations that the module manager offers are adding and removing of modules and dependencies, setting attribute values, registration of event listeners and registration of module predicates.

The module manager may also contain any kind of generic graph manipulation algorithms for the benefit of IDE actions. Operations like transitive closures of dependencies, reachability analysis, inversion, clustering, coloring and exports to graph visualizations are obvious candidates for inclusion in the module manager. Keeping the processing of these data as well as the data themselves as close as possible to the module manager will increase efficiency.

3 Module Predicates

When the user makes a change to a module, the module manager uses the dependencies between modules to trigger actions in response to that change. How can this be done in a language parametric way?

3.1 Domain analysis

Analysis of existing IDEs reveals that actions on modules are triggered either directly or indirectly. Direct actions are consequences of the actions of the programmer that are directly related to a specific module. A module is edited for example; in response to this change the system decides to invalidate the previous compilation and to trigger a new compile action. We call this directly influenced module the *pivot*. Every sequence of automatically triggered actions in an IDE always starts at a certain pivot.

Indirect actions are also consequences of the actions of the programmer, but the affected modules can be far away from the pivot. Only actions on the pivot or on modules that *depend on* the pivot can be triggered. However, these actions are triggered *conditionally* since actions for a certain module are not always triggered

$$
\begin{array}{lll}
P & ::= & N : C & & & & & & & \text{(predicate definition)} \\
N & ::= & <\textsf{Predicate Name}> & & & & & & & \text{(name of a predicate)} \\
A & ::= & <\textsf{Attribute Name}> & & & & & & & \text{(name of an attribute)} \\
V & ::= & <\textsf{Attribute Value}> & & & & & & & \text{(value of an attribute)} \\
C & ::= & \text{true} & \mid & \text{false} & & & & & \text{(Boolean constants)} \\
 & \mid & A = V & & & & & & & \text{(attribute value equality)} \\
 & \mid & \neg C & \mid & C \wedge C & \mid & C \vee C & \mid & C \rightarrow C & \text{(not, and, or, implies)} \\
 & \mid & \Diamond C & & & & & & & \text{(for some child } C \text{ holds)} \\
 & \mid & \Box C & & & & & & & \text{(for all children } C \text{ holds)} \\
 & \mid & (C) & & & & & & & \text{(parentheses for grouping)}
\end{array}
$$

Figure 1. The syntax of module predicates.

even if an (indirect) dependency changed. For example, a C program should be relinked if *one* of its dependencies changed, but only if it contains a main function, and only if *all* of its dependencies have compiled correctly. We conclude that:

- The triggering of actions is governed by the module dependency graph.
- The triggering of actions occurs under certain conditions.
- Conditions refer to properties (attributes) of modules.
- Conditions refer to the properties of children of modules.

We will use a simple language for expressing the conditions for triggering events. This language needs at least the Boolean operators, some operators for inspecting the attributes of modules, and some operators to refer to the children of modules. The idea is to evaluate these conditions *in every module*, and send an event to the IDE when the value of a condition changes. We will label each condition with a name in order to be able to identify it, and call it a *module predicate*.

The effect is that actions will be triggered automatically, in a cascading effect that starts at the pivotal change in the module dependency graph, and ends when all module predicates have been reevaluated. Note how this method of automatically triggering actions is a generalization of build tools like “make”. Those tools trigger build actions (mainly) on a set of fixed (built-in) conditions, e.g., a file being out-of-date. In our system, the conditions are programmable. Furthermore, in make-like tools dependencies and actions are tightly coupled, since every dependency rule may have a list of actions. In our system, the way dependencies are used is programmable, because a module predicate may refer to the attributes of children of modules in several ways.

3.2 Syntax and semantics of module predicates

The syntax of module predicates is defined in Figure 1. A predicate declaration consists of a predicate name N followed by a condition C. We assume that disjoint sets of predicate and attribute names are used and that predicate names are only used once. A condition may consist of true and false, tests for the value of attributes ($A = V$), the Boolean operators ($\vee, \wedge, \neg, \rightarrow$), and operators to express conditions on the children of a module that have to hold for some child ($\Diamond$) or for all children ($\Box$).

The operational semantics of the conditions is defined as follows. Each condition is evaluated for every module M. Every module has an attribute environment E that maps attribute names to attribute values, and a set of children K. The notation we use is M_K^E. An evaluation function eval reduces a condition to either true or false. It defines an operational semantics for the standard Boolean conditions (which we leave out for brevity), and an operational semantics for the conditions $A = V$, $\Box C$ and $\Diamond C$:

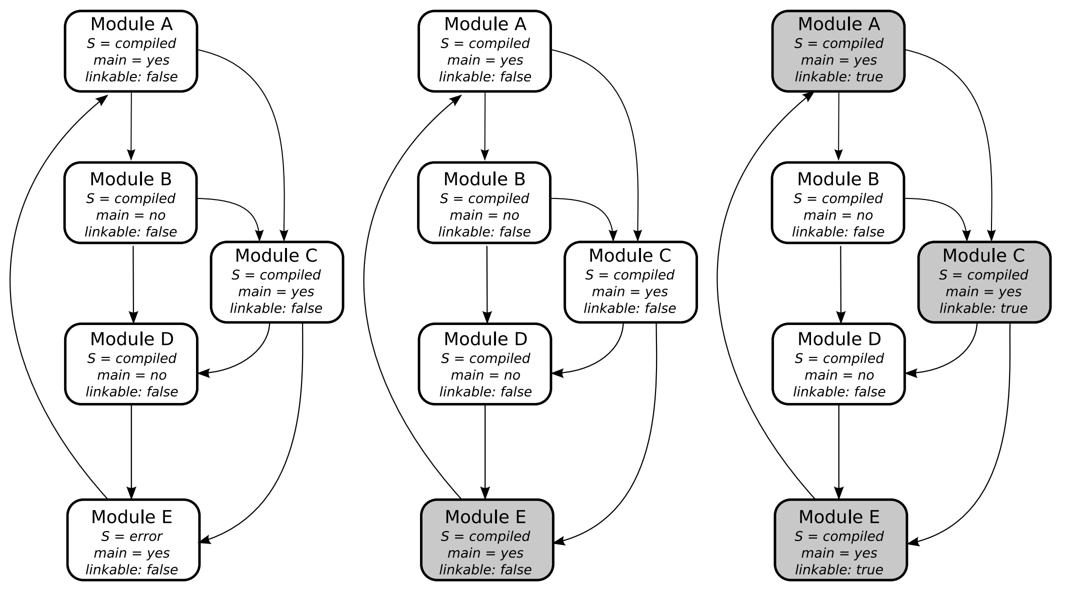
$$\begin{aligned}
\mathrm{eval}(M_K^E, A = V) &= \mathrm{true} \ \textbf{iff} \ \mathrm{equals}(\mathrm{lookup}(A, E), V) \\
\mathrm{eval}(M_K^E, \Box C) &= \mathrm{true} \ \textbf{iff} \ \forall k \in K \ \mathrm{eval}(k, C \wedge \Box C) \\
\mathrm{eval}(M_K^E, \Diamond C) &= \mathrm{true} \ \textbf{iff} \ \exists k \in K : \mathrm{eval}(k, C \vee \Diamond C)
\end{aligned}$$

Evaluating an attribute value equality amounts to a lookup of the attribute's value in the module's environment and comparing it with the given value V.

Evaluating a condition containing the $\Box$ or $\Diamond$ operator leads to the recursive application of the given condition C to the children of the current module, but evaluation differs in the way the result is aggregated. For $\Box$, the condition must succeed for all children. For $\Diamond$, the condition must succeed in for least one of the children. Note that evaluating $\Box$ and $\Diamond$ implies computing the transitive closure of the child relation among modules and that this definition of eval does not terminate on cyclic dependency graphs. A terminating definition of eval can be obtained easily by adding a cache for already visited modules for $\Box$ and $\Diamond$, but we left this detail out for clarity. Otherwise this definition terminates because it is a recursion over a finite expression tree, and no updates are done in the module environments while eval is computed. We will present a terminating (incremental) evaluation algorithm in Section 4.

The function eval is a rephrasing of the definition of the satisfaction relation of a *K4 modal logic* [2] with attribute equalities as propositions. There are several satisfiability checkers for this logic available [12, 7].

The definition of the operators $\Box$ and $\Diamond$ resembles tree traversal mechanisms, such as found in ELAN [3], ASF+SDF [4], Stratego [15], JJTraveler [16] and Strafunski [11]. However, since we are in the domain of modal logic and not in the domain of either functional programming or term rewriting, the resemblance is rather coincidental. The main difference to note is that the other operators of the language do not perform arbitrary computation but compute truth values using Boolean operators, which is at a higher level of abstraction. Another difference is that these logic operators operate on (possibly circular) graphs instead of trees.



(a) Initial state (b) Manual update (c) Predicate updates

Figure 2. Automatically updating module predicates after a manual attribute update.

3.3 Examples

The following examples illustrate how certain properties of modules can be described by a combination of attributes and predicates. In these examples we use attribute names *S* for "module state" and *T* for "module type". They serve to show the flexibility of these rules since many different kinds of action triggering policies can be expressed using this simple formalism.

$$\begin{array}{rl} \textit{erroneous}: & \Diamond(S = \textit{parse-error}) \\ \textit{linkable}: & S = \textit{compiled} \wedge \textit{main} = \textit{yes} \wedge \Box(S = \textit{compiled}) \\ \textit{not-exec}: & S = \textit{error} \vee \Diamond(S = \textit{error}) \\ \textit{package-modified}: & T = \textit{package} \wedge \Diamond(T = \textit{program} \wedge S = modified) \end{array}$$

Predicate *erroneous* flags a module as erroneous when one of its children has a parse error. An action that could be triggered when the value of this predicate changes is a user-interface action that disables certain menu options such as, for instance, executing the module. Predicate *linkable* computes whether a certain module may be linked to a runnable program. If it is a compiled main module and all of its children are compiled, then an action may be triggered to link the program. Predicate *not-exec* performs a similar computation: the corresponding module is not executable when the module itself or some of its children are in an error state. Finally, predicate *package-modified* computes that a package can be marked as modified if there is one program in its dependencies that is modified. Such a change of value of this predicate could trigger a decoration in a package explorer view that is part of the IDE.

To show how the evaluation of module predicates uses dependency relations we take, for example, the predicate *linkable* and show the update process in a few steps. We have a cyclic dependency graph as shown in Figure 2(a). The initial state is consistent; module E has $S = error$, so there is no module for which all dependencies have $S = compiled$. After a manual update in module E, its value for S changes to *compiled*, see Figure 2(b). This triggers an update of all predicates in all modules. Figure 2(c) shows that in three modules the value of *linkable* changes from false to true. In this example, the module manager will trigger actions right after the initial manual update, and then also for each separate change in valuation of a predicate in a specific module. A linker action could listen to these events and start the linking process for all three main modules.

4 Implementation

In the previous sections we have presented a high-level design of a language parametric module manager, including a data structure and syntax and semantics for module predicates. This section details some of the engineering trade-offs that are necessary to obtain an open (**R3**) and scalable (**R4**) implementation of this design.

4.1 Openness via language independent middleware

A language parametric module manager should easily allow any kind and amount of IDE extensions (**R3**). This means that many different kinds of components should be able to react to module events. This enables rapid prototyping and evolution of IDEs by reusing third-party components, by incrementally adding new components, and by gradually replacing prototype components.

Third-party components can be written in *any* programming language, but there is even a case for developing heterogenous components in-house: prototypes are usually more easily implemented in scripting languages, while the eventual product may be developed in a compiled language.

We use the TOOLBUS *component coordination architecture* to support this [1]. In a TOOLBUS-based design all "computation" is done in tools that connect via IP sockets to a software bus and all "coordination" is done via a script that describes the behaviour of this bus. As such, the tools can be written in any language and can be connected to the bus being totally oblivious from each other's existence. TOOLBUS coordination scripts can express all kinds of collaboration protocols between tools on a high level of abstraction. For example, it is easy to express synchronous and asynchronous communication, broadcasts, and locking. We use these features to construct a generic communication protocol between the module manager and an arbitrary number of tools:

- Attribute/predicate change events are broadcasted asynchronously to listeners.
- Reads and updates of the attributed module dependency graph are done synchronously.

- Reads and updates are guarded by a lock mechanism to rule out race conditions.

Tools may anonymously register as listeners. This partially implements requirement **R3** on openness. Openness can be improved further by allowing the module manager to anonymously register an arbitrary amount of module predicates at initialization time. After initialization, the module manager may present the predicates to a K4 modal logic solver in order to compute non-satisfiability and tautology. This is necessary only during development of an IDE (debugging mode). When the IDE is finished and released the set of module predicates will not change anymore.

4.2 Scalability by incremental module predicate evaluation

Our implementation should scale to *large projects* (**R4**). Large projects have large module dependency graphs that frequently contain cyclic dependencies. A straightforward implementation of the semantics of module predicates that was presented in Section 3 would visit all nodes several times, after every single update of an attribute. A small experiment showed immediately that the performance would be too low. In this section we therefore present an incremental algorithm for the efficient evaluation of module predicates.

In Section 3 we have explained that when the truth value of a module predicate changes, an event must trigger all registered listeners. The truth value may change due to a change in attribute values of a pivot module, or due to a change in the configuration of the module dependency graph. A single change in the pivot module may have as effect that many module predicates change value, triggering many actions. An efficient implementation of a module predicate evaluator should at least recalculate the truth values of all predicates that indeed have changed (i.e., the implementation should be *correct*), while it should avoid waisting time on calculating module predicates that will certainly not change (i.e., the implementation should be *incremental*).

Algorithm 1 shows an incremental predicate evaluation algorithm in pseudocode. The evaluation is started by the procedure UPDATEATTRIBUTE that initiates the value change of an attribute in the pivot module. The recursive procedure EVALUATEPREDICATES recalculates the values of all predicates that are directly or indirectly dependent on the value of the changed attribute. Note that the previous value of each module predicate *pred* is maintained in the module environment as *pred*.name thus enabling the detection of value changes with respect to the current value of *pred*.condition. The function EVALUATECONDITION computes the value of a given condition by recurring over its structure. The algorithm starts at the pivot, evaluates all module predicates, and works its way *up* the module dependency graph detecting the other modules that are affected.

We do not show the definitions of GETDEPENDENTPREDICATES (gives all predicates that depend on a certain attribute), NOTIFYLISTENERS (informs the outside world about value changes of attributes or predicates), and GETTRANSITIVECHILDREN (yields all direct and indirect children of a module). For brevity,

Algorithm 1 Incremental evaluation of predicates

```
procedure UPDATEATTRIBUTE(module, attr, value)
    global visited $\leftarrow \varnothing$
    value' $\leftarrow$ module(attr)
    if value $\neq$ value' then
        module(attr) $\leftarrow$ value
        NOTIFYLISTENERS(module, attr, value, value')
        EVALUATEPREDICATES(module, attr)

procedure EVALUATEPREDICATES(module, attr)
    if module $\notin$ visited then
        global visited $\leftarrow$ visited $\cup$ {module}
        Predicates $\leftarrow$ GETDEPENDENTPREDICATES(attr)
        for all pred $\in$ Predicates do
            value $\leftarrow$ EVALUATECONDITION(module, pred. condition)
            value' $\leftarrow$ module(pred. name)
            if value $\neq$ value' then
                module(pred. name) $\leftarrow$ value'
                NOTIFYLISTENERS(module, pred, value, value')
        for all parent $\in$ PARENTS(module) do
            EVALUATEPREDICATES(parent, attr)

function EVALUATECONDITION(module,condition)
    switch condition
        case $\Box x$:
            children $\leftarrow$ GETTRANSITIVECHILDREN(module)
            for all child $\in$ children do
                if $\neg$ EVALUATECONDITION(child, x) then return false
            return true
        case $\Diamond x$:
            children $\leftarrow$ GETTRANSITIVECHILDREN(module)
            for all child $\in$ children do
                if EVALUATECONDITION(child, x) then return true
            return false
        case attr = value:
            return (module(attr) equals value)
        case ...:
            evaluate simple boolean expressions
```

we only show the recalculation of predicates that is initiated by the update of an attribute value. When the structure of the dependency graph is changed, a similar recalculation is done.

EVALUATEPREDICATES terminates because every node is visited only once, due to the use of a global worklist. EVALUATECONDITION terminates because conditions are finitely deep, and it does not traverse the dependency graph. Instead it uses GETTRANSITIVECHILDREN, which uses a precomputed transitive closure of the dependency graph. Since $\Box$ and $\Diamond$ are transitively closed (see Section 3) this is a correct implementation. Note that we found that precomputing and caching the transitive closure saves time, since attribute updates are more frequent than adding and removing dependencies. The gain in efficiency, as compared to a naïve implementation as presented in Section 3 is caused by precomputing and caching the transitive closure of the module dependency graph and by evaluating the values of predicates in dependent modules only.

After this sketch of predicate evaluation, it is useful to understand the difference between our predicate evaluation method and conventional attribute evaluation algorithms as used in attribute grammars [14]. Attribute grammar systems take an

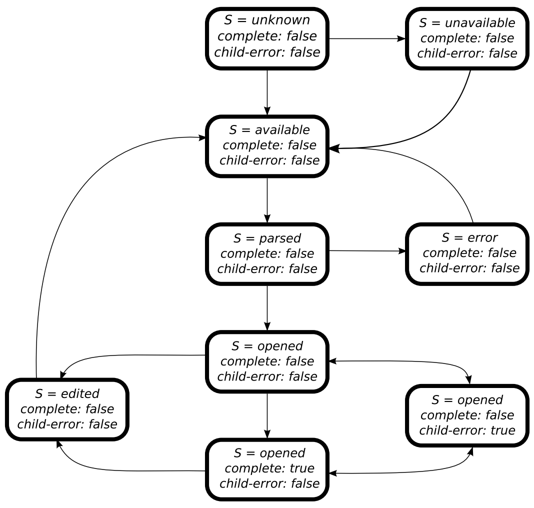


Figure 3. State transition diagram for the SDF part of an ASF+SDF module

attributed abstract syntax tree as point of departure. Attributes may be inherited (their value is propagated from root to leaves) or synthesized (their value is propagated from leaves to root). At each node, attribute equations determine the dependencies between attributes. These attribute equations induce dependencies between attributes. In most attribute grammar systems, these dependencies may not contain cycles. However, due to the existence of two types of attributes it may require several tree traversals before the computation of attribute values terminates. This is certainly true for strongly non-circular attribute grammars [10], an important subclass of attribute grammars that is widely used in practice.

Our starting point is the (possibly circular) module dependency graph. Module attributes get their value from the outside world and cannot be changed by internal computation. The operators $\Box$ and $\Diamond$ already take the transitive closure of their children into account. Therefore, predicate values can be computed in one pass over the module dependency graph. This computation always terminates.

5 Case studies

The module manager as described in the previous sections has been applied successfully in the construction of IDEs for ASF+SDF and Java. In this section we focus on the use of the module manager in these IDEs.

5.1 ASF+SDF *Meta-Environment*

The ASF+SDF Meta-Environment uses the module manager to keep module states up to date and to store other information such as graph properties, paths, and module names. Since ASF+SDF modules can introduce user-defined syntax it is helpful to treat the SDF part of a module and the ASF part separately. The remainder of this section describes the use of the module manager in ASF+SDF focusing on its use for SDF.

An SDF module can be in one of several states. The state diagram in Figure 3 describes the transitions between these states. The transitions themselves are handled by a TOOLBUS script as explained earlier. Once a module's state becomes *opened*, it is possible for the module manager to evaluate the *complete* or *child-error* predicates.

$$\begin{aligned} \textit{complete} : &\ S = \textit{opened} \land \Box(S = \textit{opened}) \\ \textit{child-error} : &\ \neg(S = \textit{error}) \land \Diamond(S = \textit{error}) \end{aligned}$$

The *complete* predicate is only true when a module and all its children are *opened*. This indicates that the module has been parsed correctly and that all of its dependencies are free of errors. When a module's state is *complete*, an action is triggered that starts the parsing process of the ASF part of the module. Since the ASF part of a module depends on the SDF part, the ASF part can only be parsed when the SDF part is *complete*. The *child-error* predicate is only true when one or more of a module's dependencies fail to parse correctly. This predicate is used as a state value in the IDE. To avoid a module getting the *child-error* state when already having the *error* state a self-check on the *error* state has been added.

5.2 Java IDE

The Java IDE is a prototype IDE for Java that uses the module manager to keep track of the same module states as for the ASF+SDF Meta-Environment , but also propagates errors and warnings through the package structure.

The import structure of Java files is very similar to the import structure of SDF files and therefore the state attribute used in the ASF+SDF Meta-Environment is reused. This also means that we can reuse some of the predicates used in the ASF+SDF Meta-Environment .

Apart from the import graph the module manager is provided with a package dependency graph. The modules of this graph consist of the segments of the package name and have Java files as leafs. We introduce the predicates *package-error*, *package-warning* and *package-modified* to describe the desired behaviour of the Java IDE:

$$\begin{aligned} \textit{package-error} : &\ \Diamond(S = \textit{error}) \\ \textit{package-warning} : &\ \neg(\Diamond(S = \textit{error})) \land \Diamond(\textit{warning} = \textit{yes}) \\ \textit{package-modified} : &\ \Diamond(\textit{vcs} = \textit{modified}) \end{aligned}$$

The *package-error* predicate is true when one or more of a module's dependencies have an *error* state. Since package segments do not have state it is not needed to have a self-check on the *error* state which is needed in case of the import dependency graph. The *class-warning* predicate is only true when one or more of a module's children have warnings. Furthermore, it can only be true if none of its children has the *error* state. The *package-modified* has been added to indicate that

	ASF+SDF Meta-Environment	**Java IDE**
Nr. of modules	75	420 (192 are libraries)
Nr. of rules involved	2	5
Modules evaluated	69	154
Time	7 msec.	265 msec.

Table 1
Performance statistics

files are modified according to the version control system. This predicate is true when one or more of a module's children are modified.

Since Java development depends strongly on package structure, the addition of package predicates is essential for a Java IDE when editing Java source modules.

5.3 Analysis

Both case studies have been carried out loading a medium-sized application in the IDE. In the ASF+SDF Meta-Environment case we used the sources of the SDF normalizer specification consisting of 75 modules. For the Java IDE case we used the source code of JSPWiki [9] which consists of nearly 38,000 lines of Java code in 228 source files and 192 libraries. In both IDEs a pivot module has been chosen in such a way that as much modules as possible were influenced by it's changes. The profile run is done by editing the pivot module and causing an error. This error propagates through the import graph, evaluating all predicates and finally evaluating *child-error* to true for all dependent modules. Only a part of all available modules will be influenced by the pivot.

Profiling these scenarios indicates that the evaluation algorithm requires a quarter of a second to compute the effects of a change in a medium-sized application. Table 1 shows the results for both case studies. We used a 2.2 GHz CPU with 1 Gb of main memory.

6 Conclusions

The module manager in previous versions of the Meta-Environment was implemented entirely in, partly language-specific, TOOLBUS scripts. The approach described in this paper is completely generic, improves the response time for state changes and reduces the size and complexity of the implementation.

The proposed module manager is fully language parametric and allows expressing module semantics in a suprisingly concise way. Module predicates can be used to propagate information through the module dependency graph. This information is IDE-specific and can be used to give feedback to the user or trigger other actions. After a change in one of the modules due to editing, module predicates can be recomputed very efficiently. Based on this positive experience we will further explore the application of this approach to other languages and IDE-features.

Acknowledgments

Jan van Eijck pointed out the similarity between our attribute evaluation mechanism and K4 modal logic; this enabled the reuse of an existing satisfiability checker.

References

[1] Bergstra, J. A. and P. Klint, *The discrete time ToolBus – a software coordination architecture*, Science of Computer Programming **31** (1998), pp. 205–229.

[2] Blackburn, P., M. de Rijke and Y. Venema, "Modal logic," Cambridge University Press, New York, NY, USA, 2001.

[3] Borovansky, P., C. Kirchner, H. Kirchner, P.-E. Moreau and M. Vittek, *ELAN: A logical framework based on computational systems*, in: J. Meseguer, editor, *RWLW96, First International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science **4** (1996), pp. 35–50.

[4] van den Brand, M. G. J., A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser and J. Visser, *The ASF+SDF Meta-Environment: a component-based language development environment*, in: R. Wilhelm, editor, *Compiler Construction (CC '01)*, Lecture Notes in Computer Science **2027** (2001), pp. 365–370.

[5] van Deursen, A., P. Klint and J. Visser, *Domain-specific languages: An annotated bibliography.*, ACM SIGPLAN Notices **35** (2000), pp. 26–36.

[6] Eclipse Foundation, *The Eclipse tool platform*, See: `http://www.eclipse.org` (2004).

[7] Fauthoux, D., *Lotrec - a general tableaux theorem prover in Java*, See: `http://www.irit.fr/Lotrec` (1999).

[8] Feldman, S. I., *Make - a program for maintaining computer programs*, Software: Practice and Experience **9** (1979), pp. 255–265.

[9] Jalkanen, J., *JSPWiki*, See `http://www.jspwiki.org` (2001).

[10] Jourdan, M., *Strongly non-circular attribute grammars and their recursive evaluation*, in: *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction* (1984), pp. 81–93.

[11] Lämmel, R. and J. Visser, *A Strafunski application letter*, in: V. Dahl and P. Wadler, editors, *Proceedings of Practical Aspects of Declarative Programming (PADL'03)*, Lecture Notes in Computer Science **2562** (2003), pp. 357–375.

[12] Le Berre, D., *A satisfiability library for Java*, See: `http://www.sat4j.org` (2004).

[13] Mernik, M., J. Heering and A. M. Sloane, *When and how to develop domain-specific languages*, ACM Computer Surveys **37** (2005), pp. 316–344.

[14] Paakki, J., *Attribute grammar paradigms — a high-level methodology in language implementation*, ACM Computing Surveys **27** (1995), pp. 196–255.

[15] Visser, E., *Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9*, in: C. Lengauer et al., editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science **3016**, Springer-Verlag, 2004 pp. 216–238.

[16] Visser, J., *Visitor combination and traversal control*, in: *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and publications* (2001), pp. 270–282.