VDM Modules

Yves Ledru[1], Marie-Laure Potet[1], and Rémy Sanlaville[2]

[1] Laboratoire Logiciels, Systèmes Réseaux - IMAG
B.P. 72 - F-38402 - Saint Martin d'Hères Cedex - France
{Yves.Ledru, Marie-Laure.Potet}@imag.fr
[2] Dassault Systèmes
9, quai Marcel Dassault, B.P. 310 - F-92156 - Suresnes Cedex - France
remy_sanlaville@ds-fr.com

Abstract. This paper evaluates the modular constructs proposed by the IFAD VDM tools. First, several case studies are presented where modules have been used for structure and refinement purposes. Then, these constructs are compared to the ones of the B method, which have been used successfully on non trivial industrial development. Finally, we study how the capabilities of VDM modules are able to match recently proposed component models like CORBA, Java Beans or ActiveX controls.

1 Introduction

Modules play several roles in specifications.

- They help structure complex systems when specifications scale up. Here, they simply transpose programming language constructs, like the Ada packages, to the writing of specifications.
- This similarity with programming languages especially makes sense when considering executable specifications where modules can be used as compilation units.
- In the context of model-based languages, modules can be seen as a concrete way to encapsulate the elements of an abstract machine (state variables and operations) or abstract data type (types and functions).
- Specification languages often include a notion of refinement. Modules can then be used to distinguish between several layers of abstraction in a refinement lattice.
- Finally, a recent trend in software engineering has fostered the component-based approach to software development. Modules can there be seen as the notion of component proposed by the specification language.

In the VDM community, the standard VDM-SL definition [1] does not provide a standard notion of module, it only points out interesting approaches to this problem. As a result, the modular construct introduced by IFAD [2] is a de facto standard for VDM.

This paper reports on experiments and reflections around this construct. Sect. 2 reports on the use of modules in several case studies. In Sect. 3, we report on the modular constructs offered by the B method. Then, we compare the VDM capabilities to recent evolutions of component based software engineering (Sect. 4). Finally, we draw the conclusions of this survey (Sect. 5) but the reader should expect more problems than solutions...

2 Feedback from Three Experiments

2.1 The Steam Boiler Case Study

The steam boiler case study was an attempt to compare a vast range of specification languages on a common problem. In [3], we have proposed a VDM specification of the problem. Our approach was to ensure requirements traceability. Therefore we started from an abstract specification which featured the major properties expected from the boiler, and gradually included details in more concrete specifications.

This specification effort ended up with three layers of abstraction, as shown in Fig. 1.

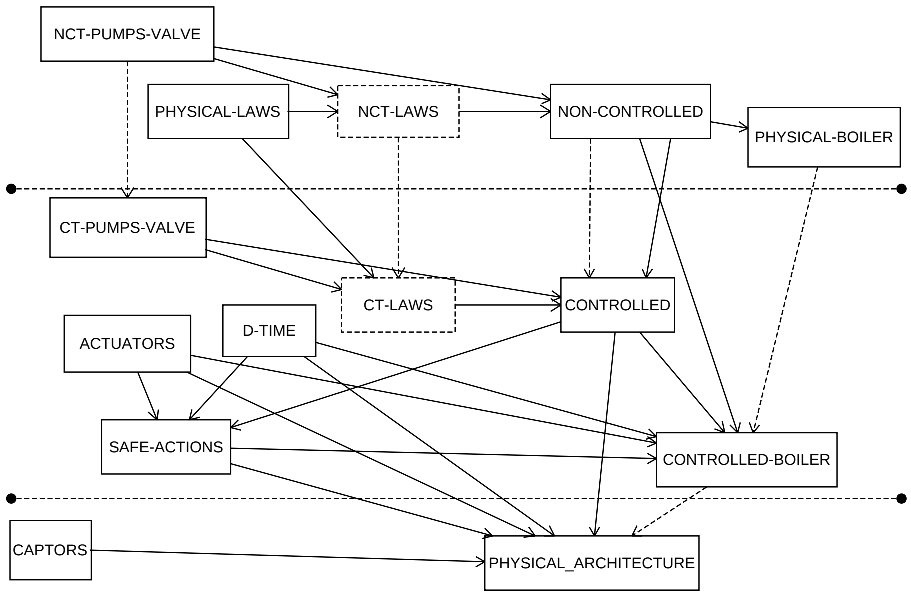

Fig. 1. Layers of abstraction in the steam boiler specification

- The upper layer describes the behaviour of an uncontrolled boiler, i.e. it encapsulates the physical laws that rule the evolution of temperature and

water level in the boiler. In other words, it describes all sensible evolutions of this physical system.
– The second layer introduces the notion of an abstract controller and expresses the expected behaviour of the boiler under control. This specification takes the global approach of considering a single system boiler+controller. It is a refinement of the first layer because the controller only allows a restricted set of the possible evolutions of the physical system.
– The lower layer decomposes the single system of layer two into three parts: the boiler, the controller, and the captors and actuators that lie at their interface. This distribution into separate units is a second refinement of the system.

The Use of VDM Modules The first version of this specification included 50 pages of commented VDM specifications. Hence, its size required the use of structuring primitives. We used the importation facility to assemble modules, and also used the parameterization facilities (e.g. the dashed boxes `CT-LAWS` and `NCT-LAWS` are produced by instantiations of the physical laws with controlled and uncontrolled versions of the specifications of the pumps).

The dashed arrows in the diagram correspond to refinement links between modules. These refinements can not be expressed by the IFAD constructs and were only documented in the associated text.

The decomposition of this system into modules resulted from a trade-off between the capabilities of the modular constructs (which allow the exportation of types, values and functions, but not states) and the attempt to follow some cohesion criteria: definition of abstract data types, distinguishing between layers of abstraction, reusing common parts like the physical laws, or encapsulating physical entities like captors and actuators.

Lessons Learned The IFAD modular constructs are inspired from programming language import/export mechanisms. This makes them easy to use which favours the construction of structured specifications. Nevertheless, they do not allow to re-export imported or instantiated constructs. This lack of transitivity makes the modular structure more complex and encourages redefinition of imported constructs to re-export them.

Also, the fact that these constructs were inspired by programming languages makes them less specific to structure specifications. In particular, the notion of refinement relation between modules is not expressed, which was perceived as a serious drawback in our experiment. Also, state definition can not be exported; this means that its definition must be duplicated, with classical maintanability problems, if it appears in two different levels of abstraction (e.g. parts of `PHYSICAL-BOILER` can not be reused in `CONTROLLED-BOILER`).

Finally, we would also have taken benefit of a parallel construct to express the fact that the boiler, controller, captors and actuators were communicating entities. Introducing a notion of thread or task would enrich the structuring primitives of the language as well as its expressive power.

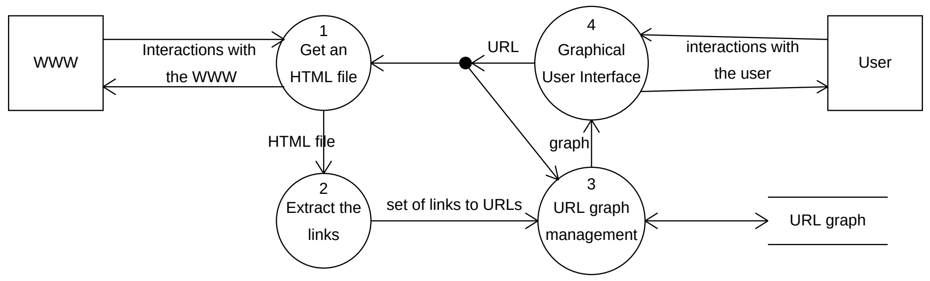

Fig. 2. A functional view of VG

2.2 A WWW Graph Visualisation System: VG

VG is a graph visualisation system which interactively displays the graph of URL links of a subset of the World Wide Web (Fig. 4).

This system is based on the integration of several components (see Fig. 2).

- A communication component (1) which reuses an implementation of the http protocol.
- A component which extracts the URL links from the HTML files (2). This component is a lexical analyser based on the lex generator.
- A graph management component (3) which stores the graph of URL links between the pages.
- A graphical user interface (4) based on the daVinci graph visualisation tool.

The graph management component was thus the only component developed in VDM, using the code generation facilities of the IFAD tools. The other components were off-the-shelf components or generated by standard tools like Lex.

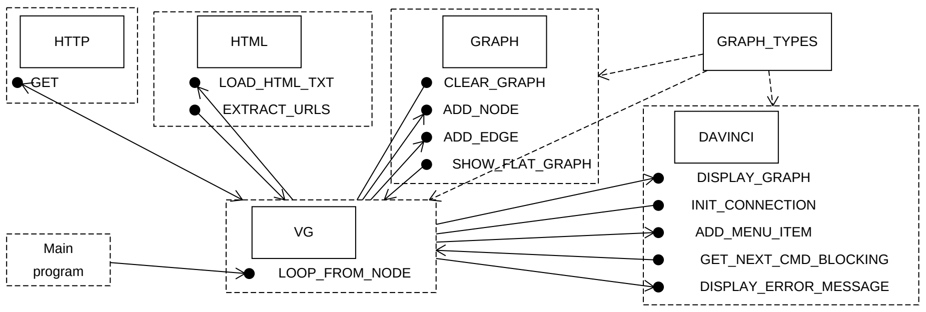

Fig. 3. Abstract machines of VG

In this experiment, we felt interesting to investigate the ability of VDM to integrate these components. Therefore, components 1, 2 and 4 were encapsulated into VDM modules as abstract machines (Fig. 3). In this specification, a component (VG) acts as a central controller which rules the data and control flows between the abstract machines.

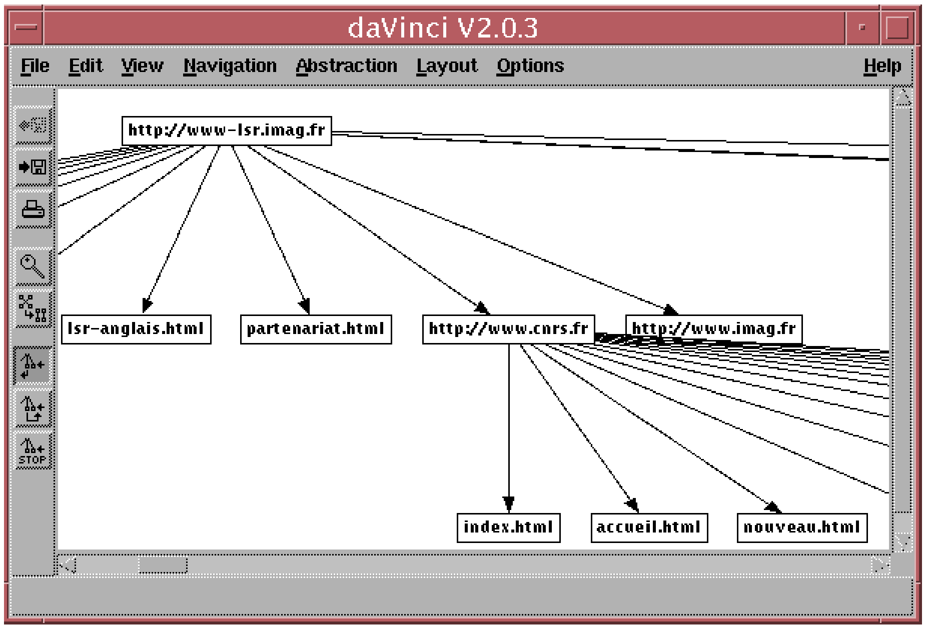

Fig. 4. The user interface of VG is based on daVinci

Lessons Learned The good news is that we were able to encapsulate very diverse components (C and lex programs, Unix executable) into VDM abstract machines and could use this paradigm as basic construct for their integration. Nevertheless, the specification of the central controller (VG) is poor: it corresponds to an imperative program. This imperative VDM specification is a low level description. In [4] we proposed the use of an Architecture Description Language which result in a higher level description of these interactions between abstract machines.

Once again we experimented difficulties with the fact that a VDM module does not export its state. As a result, it is not possible to directly access variables outside a module even to evaluate the satisfaction of a pre-condition. Also, this makes it difficult to restructure a growing module into submodules: any operation that needs to access a state variable must be specified in the module!

Finally, we noticed that IFAD modules don't separate interface from contents. When executable specifications are used, it is sensible to restrict access to the executable parts (hidden inside the module) and only expose their specification (signature, but also pre- and post-conditions) to the users of the module. Refinement can bring a solution to this problem: the body of a module is a refinement of its interface. Adding a refinement primitive to the VDM modules would allow to provide interface modules, which only include specifications, and their concrete executable counterpart.

2.3 An Indexing System

The last case study corresponds to a simple exercise we conducted in our VDM courses. It is an indexing program which produces the index of words appearing in an input text. It is made up of 6 modules (Fig. 5 and 6) which correspond to either abstract machines or abstract data types.

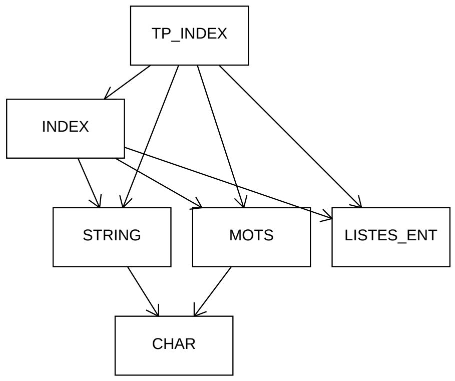

Fig. 5. Modular structure of the indexing system

Module	Nature	Contents	Used
`TP_INDEX`	Abstract Machine	4 operations, no var	1 operation
`INDEX`	Abstract Machine	11 operations, 1 var	6 operations
`LISTE_ENT`	Abstract Data Type	11 functions	5 functions
`MOTS`	Abstract Data Type	2 functions	2 functions
`STRING`	Abstract Data Type	2 functions	2 functions
`CHAR`	Abstract Data Type	2 functions	2 functions

Fig. 6. Module contents

Lessons Learned A minor problem with this case study was that some modules are actually imported but not used (e.g. `STRING` in `TP_INDEX`). This is a classical problem with modules (also in programming languages): one tends to import and export more than what is effectively used, and it results in articificial coupling.

Once again, we experimented a problem with growing abstract machines. `INDEX` features a strong common coupling around a single variable, so that 11 operations which refer to this variable must be located in this module.

The following examples illustrate the problems induced by the lack of state exportation primitives. Module `DECR` stores a single variable `x` and its invariant requires x to be greater than zero. A single operation `dec` decreases this variable. Obviously, it has the precondition that `x` must be greater than one before calling the operation. A first problem appears with this module: since `x` is only visible inside the module, how can an outside module which calls `dec` be sure that the pre-condition is satisfied? This requires either some read-only access to `x` or the ability to export the pre-condition of `dec` as an operation. This second possibility is actually featured by the IFAD tools.

```
module DECR
  exports operations dec : () ==> ()
  definitions
    state xvar of
          x : int
      inv mk_xvar(x) == x > 0
      init S == S = mk_xvar(7)
      end
    operations
      dec : () ==> ()
      dec() == x := x - 1
      pre x > 1
      post x~ = x + 1
end DECR
```

Let us now try to build a layered system where module `DECR2` builds operation `dec2` on top of the imported `dec`. The modular constructs allow us to define the code of `dec2` but the pre- and post-conditions don't type-check because `x` is not visible in `DECR2`. Strange enough, we are able to program the new operation, but not to specify it...

```
module DECR2
  imports from DECR
          operations dec : () ==> ()
  definitions
    operations
      dec2 : () ==> ()
      dec2() == (DECR`dec();DECR`dec())
      pre DECR`x > 2            -- these pre- and post-conditions
      post DECR`x~ = DECR`x + 2 -- don't typecheck!
end DECR2
```

In our last example, `show_square` corresponds to a read-only operation on `x` and we would like `DECR` to only store the critical operations (i.e. the ones that can affect the state invariant). Unfortunately, both code and specification will not type-check in module `SHOWSQUARE` because both access the hidden variable `x`. Once again, a partial exportation facility for state variables would help!

```
module SHOWSQUARE
  imports from DECR all
  definitions
    operations
      show_square : () ==> int
      show_square() == return(DECR`x * DECR`x)
      pre true
      post RESULT = DECR`x ** 2
end SHOWSQUARE
```

In summary, the IFAD primitives only feature a closed or encapsulated notion of modules. Although this approach makes sense for programming languages (it is the basis of object orientation), it is less adapted to specification activities.

3 Components and Composition

The three major model-based specification languages (VDM, Z and B) provide very different composition primitives. In the last years, the B method [5] has shown its applicability to large industrial developments (1500 components in the METEOR project), so it may provide insight for evolutions of the VDM modular constructs.

In B, the basic component is the abstract machine. It is associated to an observational semantics, i.e. any abstract machine whose observable behaviour corresponds to the specification is a valid implementation of it. This allows the substitution of an abstract machine by one of its implementations. In B abstract machines (like in VDM), the major property enforced is invariant preservation on the state variables.

The following requirements are usually expected from the composition primitives:

- At programming time, an encapsulated view is favoured, i.e. variables are hidden in the abstract machine and may only be accessed through visible operations.
- At specification time, an open view is required where internal variables are visible, as shown in the previous section.
- Composition and refinement should preserve several properties of components, like invariant preservation and substitution of a component by an implementation.

B proposes three kinds of components: abstract machines, refinements, and implementation. A refinement is a machine which refines a more abstract one; an implementation is a final (executable) refinement. B attempts to follow the three requirements listed above for composition and refinement. But its specificity is related to proofs: the method and its associated tools guarantee the correctness of specification and developments by proof obligations. Discharging these obligations is not a trivial task. Therefore it is important to ensure the compositionality of proofs: composition and refinement should preserve the correctness of a majority (if not all) of the proofs.

Several composition primitives are provided to build larger specifications from smaller ones. They have been defined as a trade-off between usability and correctness preservation: a composition should not generate too many new proof obligations. To fulfill this requirement, their use is limited by several constraints on the whole development graph. For example, one machine may only be imported once in a development, or it is forbidden to include two machines which share variables in write mode.

3.1 The Includes Primitive

The `includes` primitive is close to the state schema inclusion of Z. It gives a local copy of the variables of the imported abstract machine. It is a specification primitive, and therefore is not allowed in B implementations.

In order to preserve the proofs that have been performed on the included machine, direct access to the variables may only be read-only. Modification of the variables is allowed through calls to operations of the included machines (which preserve the invariant). These calls result in additional proof obligations to guarantee that included operations are only called within their pre-condition. Also the invariant may only be strengthened in the importing machine. As a consequence, the invariant is also valid in the including component.

Unlike VDM, B allows to re-export some of the included operations through a mechanism named promotion.

3.2 The Other Primitives

The `imports` primitive is the counterpart of `includes` for implementations, it offers the encapsulated view of components instead of the open view needed for specifications. Therefore, variables may only be accessed through the visible operations of the machine, which allows the substitution of the imported specification by one of its implementations. Moreover, `imports` is an architectural primitive: it prescribes the availability of a component corresponding exactly to the imported one.

The `sees` primitive allows variable sharing between several components in read-only mode. It is available for abstract machines, refinements and implementations. In the case of implementations, the encapsulated view is mandatory and variables can only be consulted through operation calls. The correct use of this primitive mandates some constraints on the global architecture [6].

B offers a fourth primitive (`uses`) which is never used in practice.

3.3 An Evaluation

An interesting distinction is introduced in B between specifications and implementations. This allows to adopt either an open or a closed view of abstract machines. Moreover, composition may impact the final architecture: `imports` and `sees` mandate some structure while `includes` preserve architectural freedom.

Still, B is not perfect! Experience has shown that there is space for other, more flexible, primitives that would release some constraints on the final architecture while allowing to share more information between components. But these new primitives can only be available at the cost of more proof obligations. For example, the constraint not to include two machines which share variables in write mode could be released, provided it is proved that the conjunction of both machine invariants is preserved by all operations.

Finally, B favours a local view of composition and refinements where the focus is on a restricted amount of components. For some proof activities, having a more global view of the structure of the development would result into simpler proofs.

4 Component Based Software Engineering

Since the definition of VDM, the Software Engineering community has shifted from modularity concerns to the notion of "component". First, the industry has adopted the object orientation. But classes and objects proved insufficient as component primitives. Several component models have been proposed in the recent years: CORBA, Java Beans/EJB, ActiveX/COM/DCOM,... Component technologies provide modularity, but also communication infrastructure, services for distribution, persistency, serialization,... Of course, many of these aspects are orthogonal concerns for the VDM modules, but we feel that two aspects are relevant: architectural flexibility and emerging interaction paradigms.

4.1 Architectural Flexibility

The component-based approach tends to improve architectural flexibility, in order to favour reusability of components and allow evolutive maintenance of applications. Three mechanisms can be pointed out:

Introspection: in the Java beans, naming conventions on the methods of the class allow a component to expose at run-time its visible read and write variables, the set of its methods and the events it can produce. Obviously, naming conventions can be adopted in the VDM world also to ensure compatibility with these standards.
Multiple interfaces: both Java and ActiveX/COM allow the definition of multiple interfaces. Multiple interfaces are a way to achieve polymorphism in an object model at a lower cost than multiple inheritance. In Java, interfaces are

statically defined: a component is declared to implement an interface. In ActiveX/COM, a dynamic redirection mechanism is used: the `QueryInterface` method of a default interface (`IUnknown`) is called first which returns a reference to the needed interface.
The IFAD modular construct only allows a single interface. Actually, implementing an interface is a refinement construct so it could be easily introduced in VDM specifications by a light (mainly syntactical) notion of refinement.
Object Request Brokers: CORBA and COM/DCOM allow the dynamic construction of links between independent components through queries to a request broker. This actually supports two independent notions: (a) the transparent distribution of components on a network and (b) the idea that a component provides "services" to the other components and that any provider of a given service can satisfy queries to this service. We feel that the first notion (a) is orthogonal to the concerns of VDM specifications but the second one (b) may make sense because services should be specified independently of their implementation.

4.2 Interaction Paradigms

New interaction paradigms have emerged to support the flexible assembly of components. For example, the Java beans feature event-based communication, implicit invocation, and vetoable variables.

- A Java bean is a reactive component which interacts by sending events and invoking its methods at the reception of some events. This communication mechanism is actually implemented on classical procedure calls but modeling the procedure calls in VDM is not sufficient to convey the related abstract paradigm. Moreover, the reactive behaviour of the component is exhibited *during* its execution and it is difficult to express it in a specification scheme which only refers to two instants of this execution (before and after).
- This event-based communication is the support of an implicit invocation scheme where at design time, a component does not know which components it will interact with. This adds obvious flexibility to the components. Events are not broadcasted but only sent to the components who dynamically subscribed to the event.
- Finally, a bean offers the ability to access its variables in read or write mode. It also allows other components to be aware of the evolutions of some variables, i.e. these other components receive events when the variable changes, and in some cases, they may issue a veto to some state changes.

These new interaction paradigms are not or poorly supported in VDM specifications...

5 Conclusion

In this paper, we have provided three viewpoints on the modular constructs of VDM: case studies, comparison with B, comparison with component technologies. In this conclusion, we would like to point out some interesting points:

– specifications should be able to express a variety of architectural styles and mechanisms like layered architecture, event-based systems, or implicit invocation;
– our experiments showed the need for both an open and a closed view of modules, depending on the context where the formal description is used (specification or implementation);
– structure is not only needed for specifications but also for developments, i.e. composition primitives must be designed in conjunction with refinement primitives; refinement should also address multiple interface technology;
– while modules help constructing complex abstract structures, component based software engineering also requires reusability and evolution concerns;
– modularity should also influence the design of tool support, but we have not really addressed this topic in the present paper.

Although we are far from having drawn all conclusions from these three viewpoints, we believe that there is space and need for an evolution of the structuring primitives of VDM. This evolution is vital for VDM in a world where quality components and their associated composition primitives will be increasingly demanded.

References

1. D.J. Andrews, H. Bruun, B.S. Hansen, P.G. Larsen, N. Plat, et al. *Information Technology — Programming Languages, their environments and system software interfaces — Vienna Development Method-Specification Language Part 1: Base language*. ISO, 1995.
2. R. Elmstrom, P. G. Larsen, and P. B. Lassen. The IFAD VDM-SL toolbox: a practical approach to formal specifications. *ACM SIGPLAN Notices*, 29(9):77–80, 1994.
3. Y. Ledru and M.-L. Potet. A VDM specification of the steam-boiler problem. In *Steam-Boiler Case Study*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.
4. Y. Ledru and R. Sanlaville. Description d'architecture logicielle par connexion de machines abstraites. In *2e atelier AFADL*, 1998.
5. J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.
6. M.-L. Potet and Y. Rouzaud. Composition and refinement in the B-method. In *Proc. of 2nd Int. B Conference (D. Bert, ed.)*, volume 1393 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 1998.