An Integrated Formal Methods Tool-Chain and its Application to Verifying a File System Model

Miguel A. Ferreira[1] and José N. Oliveira[2]

[1] Software Improvement Group, The Netherlands
m.ferreira@sig.nl
[2] Universidade do Minho, Portugal
jno@di.uminho.pt

Abstract. Tool interoperability as a mean to achieve integration is among the main goals of the international Grand Challenge initiative. In the context of the Verifiable file system mini-challenge put forward by Rajeev Joshi and Gerard Holzmann, this paper focuses on the integration of different formal methods and tools in modelling and verifying an abstract file system inspired by the Intel® Flash File System Core. We combine high-level manual specification and proofs with current state of the art mechanical verification tools into a tool-chain which involves Alloy, VDM++ and HOL. The use of (pointfree) relation modelling provides the glue which binds these tools together.

1 Introduction

There is a healthy trend in formal methods for computer science driven by the idea of a *Grand Challenge* (GC). Hoare [19] revisited an old challenge in computer science: a verifying compiler, capable of performing extended static analysis of the programs it compiles. Hoare's paper defines a set of criteria for an international effort to drive research in computer science forward towards automatic software verification. Hoare *et al* [20] proposed that the conditions set in [19] were met, and that the time to start such a long term international research project had arrived.

The GC project is expected to *"deliver a comprehensive and unified theory of programming"*, *"prototype for a comprehensive and integrated suite of programming tools"*, and *"deliver a repository of verified software"*. [20, Section 2]

The current paper is focused on the integration of both programming and logical tools [20, Section 2.2] that aid in the verification of formally specified operations. We propose to combine different formal specification languages, and make their tool sets interoperate, to form a tool-chain supporting a development and verification life cycle process that yields checked specifications. We assume our target audience to be already using formal specification and verification techniques, thus benefiting from a structured approach to break down software complexity through design, backed up by automated verification tools. The tool-chain should fulfil the following requirements:

– promote incremental development and verification of specifications;
– be agile enough to encourage users to verify even the smallest unit of their specifications;
– be capable of producing immediate feedback to unveil problems;
– be capable of performing fully automated consistency proofs;
– be amenable to automatic code generation.

As a case study for checking the proposed tool-chain life cycle, a formal model of an abstract file system was developed [6, 7], inspired by the "mini-challenge" proposed in [24]. Although not yet covering the robustness or hardware requirements of [24], the model built in [6, 7] is realistic while following the API of the File System Layer of the architecture for flash file systems designed by Intel Corporation [4, Section 4.14]. Such a model is given in the current paper stripped of its many details so as to convey the basic idea and method rather than not so relevant technicalities.

Paper structure Sections 2 and 3 address the integration of languages and tools in an agile tool-chain. Section 4 presents the basics of our abstract modelling strategy, based on diagrams expressing model constraints. In Section 5 the abstract (pointfree) model of Section 4 is converted to Alloy, where it is model checked for the correctness of the operations. Section 6 describes the refinements to which the Alloy model is subject to so as to render it as a VDM++ executable specification. Section 7 introduces a proof system for VDM++ that uses the HOL theorem prover to discharge proof obligations. Section 8 addresses limitations of the tool-chain and a possible implementation. Finally, some concluding remarks are given in Section 10.

It is assumed that the reader has basic knowledge of the Alloy and VDM++ languages, model checking and theorem proving.

2 Tool-chain

The main motivation for the proposed tool-chain is to combine formal method tools for model checking, theorem proving, model animation, etc, in a way such that each tool is placed in the "right" step of the given life-cycle. The version of the tool-chain which has been the subject of our experimentation involves the following languages and tools.

Relational PF-notation. Following Tarski's *formalization of set theory without variables* [35], relation algebra has emerged as a language for expressing and reasoning about logical formulae in a very concise, pointfree (PF) way. References [29, 30] show how to reason about data models using PF-notation, in a typed way supported by categorial diagrams. This paper exploits the same approach by regarding PF-notation and diagrams as the starting point of the proposed verification life-cycle.

Alloy. This is a lightweight modelling language for software design developed by the Software Design Group at MIT [22]. Its foundations are first order logic and relational calculus. Alloy's lemma "everything is a relation" makes the language very simple, highly declarative, and well integrated with the relational PF-notation, as will be explained later. Alloy's tool support is provided by the Alloy Analyzer that supports both development and verification of models.

VDM. The Vienna Development Method [2] is a mature formal method whose origins go back to the IBM Vienna Laboratory in the 1970s. The use of VDM associated languages to specify and guide the development of software has been widely described in the literature [10, 11]. VDM++ [31] is a widespread VDM dialect which, compared to ISO standard VDM-SL [32], introduces object oriented and concurrency features in the language. Tool support is one of the key strengths of VDM in general. From the wide variety of tools available we single out the Overture [26] Automatic Proof System (APS) [36] and the VDMTools [12] for type checking, interpretation and code generation.

HOL. This theorem prover [16, 34] (a descendant of the LCF theorem prover) was developed with hardware verification in mind. It is an interactive proof assistant designed for higher order logic, with a vast set of ready to use theories and proof tactics. Its function definition mechanism provides termination proofs for recursive functions for free.

3 "All-in-one" strategy

To effectively build a tool-chain it is necessary to have a strategy for each component as well as for the overall set of tools. The main goal of the strategy is to provide better verification techniques for formal development of software.

Better development means that the first steps in specifying a given problem should be taken at the most abstract level possible, capturing all the key aspects of the artifact under specification. This should be followed by incremental refinement of the specification in order to obtain an executable version, that can be used to validate functional requirements with stakeholders. Once verified, the executable specification is translated to source code in some mainstream programming language. The leap from abstract specification to executable specification must allow for early detection of failing functional requirements.

Better verification means that before tackling full-fledged proofs, confidence in the specification should be gained. In this way, one avoids attempting proofs that could be demonstrated impossible by counterexamples, or that add no value to the development since they fail the user requirements.

The kind of proof which is illustrated in the remainder of this paper is known as *satisfiability* [23]: for every operation Op whose input is of type A and whose output is of type B, proof obligation (PO)

$$\forall\ a\ \cdot\ a \in A \wedge \text{pre-}Op\ a \Rightarrow \exists\ b\ \cdot\ b \in B \wedge \text{post-}Op(b, a) \tag{1}$$

should be discharged. Because $a \in A$ and $b \in B$ check for the data type invariants associated to A and B, respectively, this PO is also referred to as *invariant preservation* [23]. Since in our case all our operations are total and deterministic, the POs we have in hands are actually simpler:

$$\forall\ a\ \cdot\ a \in A \wedge \text{pre-}Op\ a \Rightarrow Op(a) \in B\ . \tag{2}$$

The following situations can take place:

1. While specifying the overall architecture of a system, several interests are at stake. Often these interests are contradictory. A well founded notation which is paradigm-, platform- and technology-independent is welcome to enable reasoning about the high level design.
2. During the design phase, several experiments are performed to assess different design options for *Op*. A *model checker* able to automatically generate counterexamples to (2) and thus suggest how to improve *Op* is welcome.
3. *Op* satisfies (2) but is semantically wrong, for it ends up not behaving according to the requirements. To prevent this situation, running the model as a prototype in an interpreter is welcome.
4. Both the model checker and the test suite above do not find any flaws. In this case, a theorem prover is welcome to mechanically check (2).
5. PO (2) is too complex for the theorem prover. In this situation, the ultimate hope is a pen-and-paper manual proof, or some kind of exercise able to decompose the too complex PO into smaller sub-proofs.

This 5-step design scenario calls for a PO discharge strategy based on, respectively:

1. A highly abstract mathematical notation, providing for agile algebraic manipulation and diagrammatic representation of data models — we have chosen the PF-transform [30] and associated calculus of binary relations.
2. A model checker for timely generation of uninterpreted, unexpected counterexamples — we have chosen Alloy for this purpose.
3. An interpreter enabling one to carry out semantically meaningful animation and testing — for this purpose we have chosen the VDMTools.
4. A theorem prover — HOL in our case, thanks to the Overture proof system.
5. A pen-and-paper proof strategy regarding POs as "mathematical objects" which can be calculated upon. For this stage we have been using the PO calculus described in [30], where POs are represented by arrows which can be put together or decomposed into simpler ones.

This "all-in-one" strategy is depicted in Figure 1. The process starts from a highly abstract model of the architectural design of the target system, either in relational pointfree notation or directly in Alloy. Note the dashed line of the topmost box in Figure 1 (PF-notation), meaning that it is an optional stage. Although Alloy is not able to prove properties, it is very useful in finding counterexamples spotting where and why these properties fail.

After validating the design in Alloy, the model is translated to VDM++, where more detail is introduced. (Due to Alloy's notational compactness, the equivalent VDM++ specification becomes more verbose.) In the VDM++ stage it is already possible to validate all functional requirements, since the specification becomes executable. Validation can be carried out through unit tests [11, Section 9.5], combinatorial tests [27], or by interpreting (animating) the specification. Should dynamic analysis performed at VDM++ level detect any design flaw, the process goes back to the Alloy stage to suppress defective behaviour. Once the specification looks *adequate* and captures all functional requirements, the Overture APS is used to generate all the POs arising from the VDM++ model and attempt to mechanical discharge them in HOL.

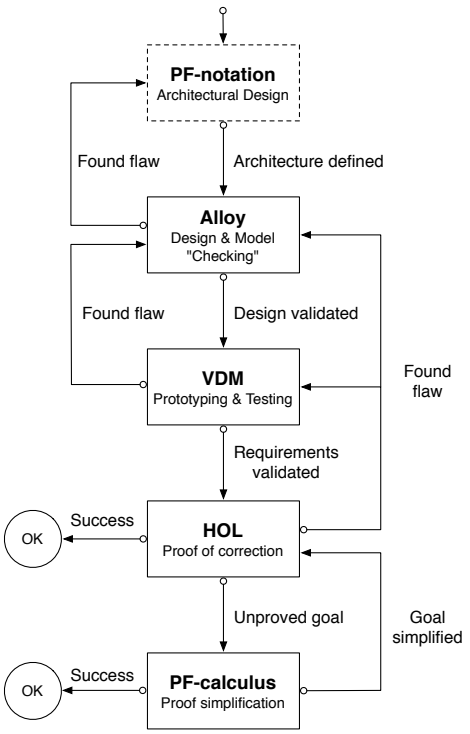


Fig. 1. Tool-chain operation.

The last stage (pen-and-paper proof) caters for POs which HOL could not prove and Alloy could not refute: the worst scenario. The idea is to use PF-calculation at this stage, aiming at simplifying POs or dividing them into smaller goals, which are fed back to HOL.

4 Relational model of a (simplified) file system

At the highest level of abstraction, a file system model should only capture the top level relationships among its main components. Capturing the properties which constrain the system's overall state is an essential part of This exercise.The challenge is doing so in a way which helps in reasoning about operations over such constrained state. At this level, the less detail the better, as long as no key aspect is overlooked.

A very abstract relational model of a file system is presented using PF-notation. Relational point-free models are built by depicting binary relations as arrows between data types in diagrams. The diagrams have a strong formal semantics, based on category/allegory theory [14], thus ensuring the move from diagrams to symbols, back and forth.

$$Path \xleftarrow{dirName} Path$$

At such an abstract level, a file system *stores* files in a way such that their data becomes *accessible* through paths. Paths play the double role of *identifying* files and revealing the *hierarchy* under which they are stored. Following POSIX terminology, we define the relation $dirName$ that for a given path yields its parent path. This relation establishes the hierarchy of files within a file system: a file a is said to be the parent of a file b if, in the hierarchy, b lies exactly underneath a, that is, $(path\ a)dirName(path\ b)$ holds.

Just by thinking of paths one pictures a file system as an hierarchic structure, in fact a tree like structure, provided some properties of $dirName$ hold. This does not necessarily mean that a file store must be a tree structure. As long as it is possible to navigate throughout it, any structure can implement a file store. Given a file system s, its file store component $fileStore\ s$ is abstractly specified as a partial function.

$$Path \xleftarrow{(fileStore\ s)} File$$

Partial functions are often termed *simple* relations [1], and we shall use this terminology too. Simple relations are so important in our data models (as elsewhere) that we use special harpoon looking arrows to depict them in diagrams, as above. Files can be handled by applications through the file system API, provided that all applications relying on files can reach them, and the files they are using do not get moved or removed. Applications do not handle files directly, instead they do it through file handlers. It is the file system's task to manage the relation between file handlers and the corresponding file descriptors. These descriptors keep relevant run-time information about files that are open, and in use by applications.

$$FileHandle \xleftarrow{(table\ s)} FileDescriptor$$

This leads us to a file system model with two subcomponents: a file store, and an open-file table. The file system requires from the file store the ability to find a file given its path, and that the open-file table keeps track of the files requested by applications.

$$\begin{array}{ccc} FileDescriptor & \xleftarrow{table\ s} & FileHandle \\ {\scriptstyle path}\downarrow & & \\ Path & \xrightarrow[fileStore\ s]{} & File \\ {\scriptstyle dirName}\uparrow & & \\ Path & & \end{array}$$

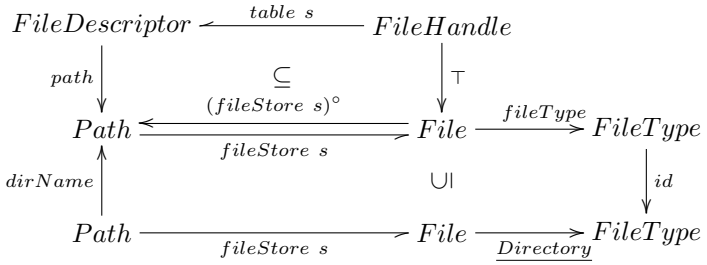
Files are the basic unit of a file system, and POSIX [21, Section 3.163] defines several types of files: regular file, directory, character special file, block special file, fifo special file, symbolic link and socket. Only regular files and directories are of interest at this topmost level of abstraction, and to distinguish these two types of file we introduce the $fileType$ relationship, cf.:

$$\begin{array}{ccccc} FileDescriptor & \xleftarrow{table\ s} & FileHandle & & \\ {\scriptstyle path}\downarrow & & & & \\ Path & \xrightarrow[fileStore\ s]{} & File & \xrightarrow[fileType]{} & FileType \\ {\scriptstyle dirName}\uparrow & & & & \\ Path & & & & \end{array}$$

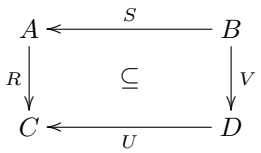
The next step in the modelling consists of "gluing" the data structures in the diagram with constraints spelling out their static semantics (data type invariants, in the VDM terminology). The following pair of constraints is easily extracted from [4]:

- **Referential integrity**: non existing files cannot be handled by applications.
- **Paths closure**: parent directories always exist and are indeed directories.

In diagrams, constraints take the shape of rectangles, each labelled by the appropriate relational inclusion symbol [3]:



The diagram above depicts the two constraints that were identified: *referential integrity* (ri) is the top rectangle, *paths closure* (pc) is the bottom rectangle. Let us explain the meaning of these diagrams. A rectangle as displayed aside depicts PF-formula $R \cdot S \subseteq U \cdot V$. Once the meaning of relational composition is spelt out, this PF-formula becomes predicate



$$\forall\ c \in C, b \in B\ \cdot (\exists\ a \in A\ \cdot cRa \wedge aSb) \implies (\exists\ d \in D\ \cdot cUd \wedge dVb)\ .$$

Should any of R, S, U, V be the top relation $\top$, the corresponding conjunct is deleted from the formula above because $y \top x$ always holds, for any choice of x and y. Should it be a converse relation, say R°, then the variables of the corresponding conjunct are swapped, because $yR^\circ x$ means the same as xRy. Finally, should it be a function f, then $y\ f\ x$ means the same as $y = f\ x$, thus cancelling quantification over y. In the particular case of f being the everywhere-k constant function $\underline{k}$, $y\ f\ x$ shrinks to $y = k$.

In this way, the rectangle picturing referential integrity — which in symbols is $path \cdot (table\ s) \subseteq (fileStore\ s)^\circ \cdot \top$ — unfolds (for all $b \in FileHandle$) into predicate:

$$(\exists\ a \in FileDescriptor\ \cdot a(table\ s)b) \implies (\exists\ d \in File\ \cdot d(FileStore\ s)(path\ a))\ .$$

Drawing constraints in this fashion, as rectangles in diagrams, allows for great notation economy while providing for the visualization of the design in a "UML-like" style. The interested reader will want to do the exercise of spelling out the predicate which is pictured by the other rectangle in the diagram.

[3] In category/allegory terminology, these rectangles are referred to as "commutative squares".

Regarding the file system API, we specify operations as arrows again. Aside we consider the two operations *open* and *create*, which open a given (regular) file and create a new (regular or directory) file, respectively. Opening files results in a new state with a new entry in the open-file table and a file handle referring to it. Creating a file only modifies the file store, by adding the new file.

$$System \times Path \xrightarrow{open} System \times FileHandle$$
$$System \times Path \times FileType \xrightarrow{create} System$$

In the next section the relational pointfree diagrammatic specification is translated to Alloy in an almost effortless exercise.

5 From PF diagrams to Alloy

Transposing the above relational specification to Alloy is almost direct, since Alloy relations are first class citizens. Still, more detail is required in the Alloy specification to more accurately specify the file system state and operations. Once the specification is transliterated to Alloy, and the Alloy Analyzer is asked to instantiate it, it will display instances where: *(a)* there are cycles in paths; *(b)* there are directories being referenced in the open-file table. Both these situations should be avoided either because they are erroneous states of the file system *(a)* or because they display undesired behaviour *(b)*. To overcome this situation, more constraints must be added to the specification, and to a certain extent more detail has to be introduced in signatures (Alloy data types). Considering the file store only an additional constraint should be enforced:

Paths structure: the *dirName* relation should be such that: (a) the root directory is its own parent; (b) it is acyclic for all paths other than the root directory (thus no links are allowed in the file system).

Regarding the open-file table, one more constraint should be enforced:

Files table: only regular files can be opened, ie. no entry in the open-file table should refer to a directory.

The two simple relations of the diagrams lead to the Alloy top-level signature aside. In the system definition, the harpoon arrow of the relational diagram becomes the *lone* (one or less) multiplicity factor. Hence simplicity is ensured.

```
sig System {
  fileStore: Path -> lone File,
  table: FileHandle -> lone FileDescriptor
}
```

We specify *dirName* as a simple and total relation on paths, thus a function from *Path* to *Path*. Both simplicity and totality of the relation are specified with the *one* multiplicity factor in the range of the relation. This means that no path has more than one parent path and that, at the same time, every path (in the relation) has a parent path.

Note the use of the *abstract* keyword in declaring the *Path* signature, meaning that there can be no instances of *Path*. Using this keyword only makes sense if one extends the signature later on. In Alloy, the extension mechanism is similar to OO-extension in the sense of inheriting the structure and properties of the extended entity. Furthermore, by extending an abstract signature one creates a partition of that signature.

```
abstract sig Path {dirName: one Path}
```

The root path is different from any other path, and this reflects the hierarchy of a file system, where the root is the topmost element. To differentiate the root from the other paths, we introduce it as an extension to *Path*. Furthermore, the root path is declared to be unique, through signature multiplicity factor *one*. The remaining paths are instances of the *FileNames* signature. Upon root path differentiation, separate properties can be specified for each type of path present in the *dirName* relation, namely:

```
one sig Root extends Path

sig FileNames extends Path {}
```

```
pred ps[] {
  Reflexive[id[Root].dirName,Root]
  Acyclic[id[FileNames].dirName,FileNames]
}
```

The *path structure* (*ps*) predicate enforces the paths structure constraint, by declaring that: *(a) dirName* is reflexive on the root path, ie. root is parent of itself; *(b) dirName* is acyclic for all other paths.

To specify the remaining constraint (*files table*) it is necessary to differentiate files by their type. Although we have already introduced file types in the relational specification, we left room for choice on this matter.

```
sig File {fileType : one FileType}
```

```
abstract sig FileType{}

one sig RegularFile, Directory extends FileType {}
```

One way to make such differentiation explicit is to partition files, as done before concerning paths. However, in this case, it is not necessary to define separate relational properties for each type of file, and therefore, it suffices to use a flag as differentiation mechanism. File types are defined as a partition composed of regular files and directories.

```
pred ft[s: System] {
  (s.table).path.(s.fileStore).fileType
    in (FileHandle -> RegularFile)
}

pred inv_System[s: System] {
  ri[s] and pc[s] and ft[s] and ps[]
}
```

The *files table* (ft) constraint predicate (above) enforces that only (regular) files can be requested by applications to read and write. (Without prejudice of directories being browsable.) The overall invariant for the system is then defined as a conjunction of the two constrains *referential integrity* (ri) and *paths closure* (pc) defined in the relational specification, and the above described constraints *files table* (ft) and *path structure* (ps).

Once the state of the system is defined we proceed to the specification of the operations. Each operation is specified as n-ary relation Op between an initial system s and a final state s', for instance:

```
pred openFile[fh': FileHandle, s',s: System, p: Path] {
  s'.fileStore = s.fileStore
  fh' !in s.table.dom
  (one fd: OpenFileInfo {
      fd !in s.table.ran and fd.path = p
      s'.table = s.table + (fh' -> fd)
  })
}
```

. The operation *openFile* does not affect the file store and produces a new entry in the open-file table. It is guarded by a precondition made of two conjuncts. The first is meant to preserve referential integrity, and the second to preserve the open-file table invariant.

```
pred pre_openFile[s: System, p: Path] {
  p in s.fileStore.dom
  p.(s.fileStore).fileType = RegularFile
}
```

With the Alloy Analyzer it is possible to start verifying this operation straight away, and we do so by first simulating and afterwards verifying. Either because the scope is too narrow, or because a predicate is a contradiction, verifying assertions will always succeed if there is no possible instantiation. To detect these situations, we make sure that the predicate can be instantiated by simulating it. Simulation with Alloy Analyzer can easily reveal problems as the instances of the model are depicted in simple (but expressive) diagrams. We have checked the *openFile* operations for satisfiability, with a scope of 10 elements[4], and found no counterexamples.

The *create* operation creates a new file, of a given type, in the file store.

```
pred create[s',s: System, p: Path, ft: FileType] {
  s'.table = s.table
  one f: File {
    f.fileType = ft
    s'.fileStore = s.fileStore + (p -> f)
  }
}
```

[4] In Alloy the state space is limited by the scope. The scope defines how many elements will be used for each top level signature. Top level signatures are those which do not extend other signatures.

The operation is guarded by a precondition again made of two conjuncts. The first prevents from creating files that already exist in the file store. The second is composed of a disjunction of two other sub-clauses. The first of these allows one to create the root directory (note that this is only possible if the file store is empty due to the first clause). The second preserves the paths closure invariant, in case a path other than the root is passed as argument.

```
pred pre_create[s: System, p: Path, ft: FileType] {
  p !in s.fileStore.dom
  ((p = Root and ft = Directory)
   or
   (p.dirName in s.fileStore.dom and
    p.dirName.(s.fileStore).fileType = Directory))
}
```

After simulation and verification, no counterexamples where found for the *create* operation, also for a scope of 10 elements.

6 From Alloy to VDM++

Model translation to VDM++ involves additional effort and increases the steepness of the learning curve. However, it helps in further refining the specification, while giving access to a comprehensive set of tools.

VDM++ translation is guided by the rules described in [7, 6]. The outcome is a sizeable VDM++ model of which we only address an example of where the abstraction level is lowered, in order for the specification to become executable.

The refinement that has greater impact in the model relates to paths. Paths in the Alloy model are so abstract that it suffices to differentiate the root and declare a relation (*dirName*) recording the path-hierarchy. There are two obvious models for paths in VDM++: either as a linear recursive data type, or as a sequence of file names.

The first option would clash with the mapping we chose to use for the file store, because it would introduce inductive reasoning (which we decided to avoid). The second option, which was chosen, allows us to avoid inductive reasoning, but introduces some more constraints. The resulting VDM++ data type that specifies paths is defined as a co-product of root and remaining paths, as in Alloy. The difference resides in the specification of the remaining paths, now sequences of tokens.

```
Path = <Root> | seq1 of token;

dirName : Path -> Path
dirName(p) ==
  cases p:
    <Root> -> <Root>,
    [-]    -> <Root>,
    others -> allButLast(p)
  end;
```

The refinement of paths introduces a new constraint preventing paths, which are sequences, from being empty. The relation that navigates through paths

(above) must also be refined according to the changes in the data type, where the alternative pattern [−] matches any singleton sequence.

Recall, from the specification of *openFile*, that the entry to be created in the open-file table should consist of a new file handle and a new file descriptor. In Alloy it was possible to declare that the file handle should not belong to the original table; in VDM++ it is necessary to operationalize this behaviour.

```
open: System * Path -> System * FileHandle
open(s,p) ==
  let newFh  = newFileHandle(dom s.table),
      entry  = { newFh |-> mk_OpenFileInfo(p) },
      table' = s.table munion entry in
  mk_(mu(s, table |-> table'),newFh)
pre p in set dom s.fileStore and
    s.fileStore(p).fileType = <RegularFile>;
```

In the above definition a new file handle is mapped to a new open-file element using the binary operator `|->`. The initial state is mutated using the *mu* operator, whereby the original *table* field is replaced by the newly created $table'$.

7 From VDM++ to HOL

For each PO arising from the specification, the Overture proof system can yield three different results:

1. the PO evaluates to *true* (discharged) — no inconsistency found;
2. the PO evaluates to $false$ — a design inconsistency exists;
3. the PO evaluates to an unproven goal — no conclusion from proof.

In the case of a discharged PO (Item 1) the life cycle is over for this particular PO. If, on the contrary, the PO evaluates to false (Item 2) then it is clear that a flaw exists in the specification and some action must be taken to correct it. At this stage the adequate corrective action depends on the kind of flaw detected. It might be the case that the proof failed because of some error introduced in one of the previous stages, Alloy or VDM++. So the process should go back to the appropriate stage to correct the specification. The last possible outcome (Item 3) might result from a proof that stopped before reaching any of the Boolean values, or from a proof that times out.

Through the Overture proof system the specification was analysed to generate the two satisfiability proof obligations for the specified operations [6]. A HOL theory was automatically translated from the VDM++ specification, and a proof script produced. (Neither the theory nor the proof script are described in this paper, due to space constraints — see [6] for details.) It followed that the proof system was able to mechanically discharge the satisfiability proof for

$$inv \xleftarrow{open} inv \tag{3}$$

but not for

$$inv \xleftarrow{create} inv \tag{4}$$

(We adopt the arrow notation of [30] for satisfiability proof obligations.)

Recall that *inv* is a conjunction of four predicates: *referential integrity*, *paths closure*, *files table* and *path structure*. The last is no longer necessary because it is ensured by the $dirName$ function once refined to the VDM level. Following the *splitting by conjunction* rule of the PO-calculus of [30, Section 15], (4) splits into:

$$ri \xleftarrow{create} inv \tag{5}$$

$$pc \xleftarrow{create} inv \tag{6}$$

$$ft \xleftarrow{create} inv \tag{7}$$

Sub-goals (5) and (7) were mechanically discharged by the proof system, whereas (6) produced an intermediate goal. Further decomposition applied to (6) branches the proof into: *(a)* the case where the argument path is the root directory; *(b)* the remainder cases. By manipulating the theorems made available to the prover for term rewriting, the two branches of (6) where interactively discharged in HOL.

The success of the proof was due to initially limiting the theorems used by the re-writing procedures. The first attempt to discharge this proof used all available theorems from the specification theory to re-write and simplify the goal. However, this approach lead to an intermediate goal whose semantics could only be perceived by inspecting every proof step to identify all relevant decisions that took place.

By not allowing the prover to use the theorems for $dirName$ and pc (paths closure invariant), and re-invoking the same proof tactic we obtained branches *(a)* and *(b)*. In this way, goal (6) was split in two sub-goals (one per branch), the theorems for $dirName$ and pc were made available for the re-writing tactics, and the remaining proof was carried out automatically by the APS.

8 Discussion

This paper presents a formal methods tool-chain that promotes tool interoperability while transforming abstract models through an iterative process of development. The tool-chain disciplines the use of different tools and techniques ranging from simulation, model checking, testing, interpretation and code generation, to mathematical proof of correctness.

For the tool-chain to be applicable in the verification of large and complex models some issues have to be addressed. First of all, not every step in the tool-chain is automated. Although the Overture APS automates the connection between VDM++ and HOL, the one between Alloy and VDM++ is still manual. First steps towards this automation have been taken in [7, 6] by defining a

set of rules to translate VDM++ data types into Alloy signatures. In the current paper similar rules are applied, however from Alloy to VDM++. We agree that the agility of the tool-chain is compromised until all steps are fully automated. Although code generation was not addressed in the paper, the tool-chain "borrows" this capability from the VDMTools.

Of the tool-chain requirements set up in Section 1, only fully automated proofs are still far from being a reality. These should eventually include those of the refinements implicit in translating from one notation (eg. Alloy) to another (VDM++). It is our intention to experiment with the presented tool-chain to verify the different refinements it promotes. Extending the verification capabilities of the tool-chain to support refinement proofs would indeed increase its usefulness and soundness. Surely, there is much work to be carried out in this respect.

With the file system case study we show how a small model can be fully verified in a multi-stage process. Stage after stage (Figure 1), more confidence is gained on the consistency of the model. Throughout this case study care was taken to independently check small units of models, by constructing the model piece by piece on a tight loop of development and verification. However, when verifying models whose development is out of the verifier control, *slicing* tools [37] are of great value, since they can isolate the smallest sub-model that accommodates some target property, operation or data type. This is another aspect which calls for automation: operation-wise manual slicing carried throughout the project [7, 6] proved to be very time-consuming.

Both the languages and principles adopted in devising the tool-chain are generically applicable to software development and verification. We therefore envisage its integration in the Overture platform in the near future. Overture includes a framework for generation of abstract syntax trees (ASTs) for languages modelled in VDM++. This framework is supported by the AST generator (AstGen) tool, which (for example) was used to generate the Overture Modelling Language (OML) AST from a VDM specification. OML AST is the pillar that supports all other Overture tools that manipulate VDM dialects. Both OML AST and surrounding tools can be automatically implemented in Java (or C++) [17]. The Overture proof system stands as an example of such automated implementation. Adding to these features, efforts are currently under way to integrate the complete Overture tool set in the Eclipse platform, where Alloy is already integrated [28]. All these conditions together with the fact that the VDM++ connection to HOL is a component of the Overture tool set, make this the most interesting option to foster the tool-chain put forward in this paper.

9 Related and future work

Verifiable file system. Since the VFS mini-challenge was put forward, contributions have been made at different levels, either focusing on verification or refinement [25, 13, 5]. Reference [25] already contemplates NAND flash memory peculiarities, such as wear levelling, erase unit reclamation, and tolerance to

power loss. More recently, new papers [33, 18] on file system formalization have become available. Theorem proving is used in [18], which follows a top down approach in formalizing a hierarchical file system. [33] reports on the bottom up verification of the UBIFS implementation for Linux.

Other file system implementations have also been mechanically verified by model checking [15, 38]. [38] found several errors in widely used file system implementations that were reported back to the respective developers. [15] analysed a concurrent model of the Linux Virtual File System, which bridges between the Linux kernel and the miscellaneous file system implementations that it supports.

Integration of formal tools. There has been a proliferation of independent languages and tools that support formal specification and verification. However, it is already possible to see the results of the effort made towards integrating these tools in development environments that are more agile and sophisticated. Good examples of such integration are Alloy4Eclipse [28], the Rodin [3] tool for Event-B and the Overture tools for VDM.

Part of the tool-chain presented in the current paper is already implemented in the Overture project, thanks to our work on the APS workflow [9]. Current efforts go into improving interoperability among Overture internal components, the VDMTools and HOL. This will hopefully produce a cross platform proof system, capable of mechanically discharging all VDM-standard POs.

On flash-level refinement. As for current work on the VFS project itself, our implementation (refinement) strategy is based on the following design principle: whatever abstract model one writes for file systems, it can be refined into diagrams of "atomic" (1NF) simple relations using *data transformation by calculation* [29]. Inspired by [33], one just has to consider a further, generic refinement step in moving towards the flash level: that of implementing every simple relation by a 4-tuple made of the relation itself, the corresponding RAM and flash indices and the *journal*. We are currently busy in proving the correctness of this refinement strategy [8].

10 Summary

The research described in this paper is intended to contribute to the GC trend while focusing on tool interoperability as a means to obtain an integrated verification tool-chain taking advantage of the capabilities of each tool in the chain.

The integration of formal language tool sets in modern development environments such as Eclipse is today a reality. We propose that communities take an extra step towards interoperability. This can be done through translators based on public ASTs, that can be distributed to developers of other communities as open source code, or binary libraries. However, the soundness of such integration still needs to be demonstrated through refinement proofs.

This paper shows how the principles of abstraction, iterative development and proof decomposition help in overcoming the difficulties implicit in verifying

complex operations on states subject to elaborate invariants. Operation can be broken down in sub-operations that are independently verified. Invariants can be factored into sub-invariants. In the case study reported in this paper [6, 7], decomposition helped in identifying properties and sub-operations that were preventing the proof system from automatically discharging the proof.

In retrospect, the improvements in verification obtained following our "single-PO, multiple-proof-technology" approach need to be balanced against the fact that the learning curve becomes steeper and steeper as new technologies are added to the system. This can only be avoided via automation and transparent integration.

References

1. R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C.A.R. Hoare, series editor.
2. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
3. J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. RODIN (Rigorous open Development Environment for Complex Systems). In *WORDS*, pages 23–26. IEEE Computer Society, 2005.
4. Intel Corporation. Intel® Flash File System Core Reference Guide. Technical report 304436-001, Intel Corporation, 2004.
5. K. Damchoom, M. Butler, and J. Abrial. Modelling and Proof of a Tree-Structured File System in Event-B and Rodin. In S. Liu, T.S.E. Maibaum, and K. Araki, editors, *ICFEM*, volume 5256 of *LNCS*, pages 25–44. Springer, 2008.
6. M. Ferreira. Verifying Intel® Flash File System Core. Master's thesis, Minho University, Jan. 2009.
7. M. Ferreira, S. Silva, and J.N. Oliveira. Verifying Intel Flash File System Core Specification. *Fourth VDM/Overture Workshop*, (CS-TR-1099), May 2008.
8. M.A. Ferreira and J.N. Oliveira. Verifying the (generic) flash memory implementation of abstract mappings, 2009. In preparation.
9. Miguel A. Ferreira. Implementing the Overture Automatic Proof System, 2009. Submitted for publication.
10. J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998.
11. J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.
12. J. Fitzgerald, P.G. Larsen, and S. Sahara. VDMTools: advances in support for formal modeling in VDM. *SIGPLAN Notices*, 43(2):3–11, 2008.
13. L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/Eves: an experiment in the verified software repository. In *ICECCS '07*, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.
14. P.J. Freyd and A. Ščedrov. *Categories, Allegories*, volume 39 of *Math. Lib.* North-Holland, 1990.
15. A. Galloway, G. Lüttgen, J.T. Mühlberg, and R. Siminiceanu. Model-checking the linux virtual file system. In N.D. Jones and M. Müller-Olm, editors, *VMCAI*, volume 5403 of *LNCS*, pages 74–88. Springer, 2009.
16. Mike Gordon. *From LCF to HOL: a short history*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

17. The VDM Tool Group. The VDM++ to Java Code Generator. Technical report, CSK Systems, January 2008.
18. W.H. Hesselink and M.I. Lali. Formalizing an Hierarchical File System, 2009. Submitted to FM 2009.
19. C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
20. T. Hoare and J. Misra. Verified Software: Theories, Tools, Experiments Vision of a Grand Challenge Project. In *VSTTE*, pages 1–18, 2005.
21. IEEE and The Open Group. Standard for information technology - POSIX®. Base Definitions, Issue 6. *IEEE Std 1003.1-2001. The Open Group Tech. Std*, 2004.
22. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Heyward Street, Cambridge, MA02142, USA, April 2006.
23. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.
24. R. Joshi and G. J. Holzmann. A Mini Challenge: Build a Verifiable Filesystem. In *VSTTE*, pages 49–56, 2005.
25. E. Kang and D. Jackson. Formal Modeling and Analysis of a Flash Filesystem in Alloy. In E. Börger, M. Butler, J.P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *LNCS*, pages 294–308. Springer, 2008.
26. P.G. Larsen, N. Batle, J. Fitzgerald, K. Lausdahl, and M. Ferreira. The Overture Initiative Integrating all VDM tools, 2009. In preparation.
27. P.G. Larsen, K. Lausdahl, and N. Batle. Combinatorial Testing for VDM++, 2009. Submitted for publication.
28. D. Leberre and F. Delorme. An eclipse plugin for the alloy4 tool. Website: `http://code.google.com/p/alloy4eclipse/`.
29. J.N. Oliveira. Transforming Data by Calculation. In *GTTSE'07*, volume 5235 of *LNCS*, pages 134–195. Springer, 2008.
30. J.N. Oliveira. Extended Static Checking by Calculation using the Pointfree Transform. In A. Bove et al., editor, *LerNet ALFA Summer School 2008*, volume 5520 of *LNCS*, pages 195–251. Springer-Verlag, 2009.
31. Larsen P.G, J.S. Fitzgerald, and S. Riddle. Practice-oriented courses in formal methods using VDM++. *Formal Asp. Comput.*, 21(3):245–257, 2009.
32. Nico Plat and Peter Gorm Larsen. An overview of the ISO/VDM-SL standard. *SIGPLAN Notices*, 27(8):76–82, 1992.
33. A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract Specification of the UBIFS File System for Flash Memory, 2009. Submitted to FM 2009.
34. K. Slind and M. Norrish. A Brief Overview of HOL4. In O.A. Mohamed, C.M., and S. Tahar, editors, *TPHOLs*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.
35. A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*. American Math. Soc., 1987. AMS Colloq. Pub., v. 41, Providence, Rhode Island.
36. S. Vermolen. Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University, Computer Science Department, 2007.
37. M. Weiser. Program slicing. In *5th Int. Conf. on Software Eng., San Diego, California*, March 1981.
38. J. Yang, P. Twohey, D.R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.