Modelling and Proof of a Tree-Structured File System in Event-B and Rodin*

Kriangsak Damchoom[1], Michael Butler[1], and Jean-Raymond Abrial[2]

[1] University of Southampton
United Kingdom
{kd06r,mjb}@ecs.soton.ac.uk
[2] ETH Zurich
Switzerland
jabrial@inf.ethz.ch

Abstract. Event-B is a formalism used for specifying and reasoning about complex discrete systems. The Rodin platform is a new tool for specification, refinement and proof in Event-B. In this paper, we present a verified model of a tree-structured file system which was carried out using Event-B and the Rodin platform. The model is focused on basic functionalities affecting the tree structure including create, copy, delete and move. This work is aimed at constructing a clear and accurate model with all proof obligations discharged. While constructing the model of a file system, we begin with an abstract model of a file system and subsequently refine it by adding more details through refinement steps. We have found that careful formulation of invariants and useful theorems that can be reused for discharging similar proof obligations make models simpler and easier to prove.

Keywords: File system, Tree structure, Refinement, Proof, Event-B, Rodin tool.

1 Introduction

Nowadays, there are many formal methods used in the area of software development together with a number of advanced theories and tools. However, more experiments in this area are still needed to be carried out in order to provide significant evidence for convincing and encouraging other users to benefit from those theories and tools, and make formal methods more accessible to software industries. We see our work as a contribution to the filestore mini-challenge proposed by Joshi and Holzmann [13]. As highlighted in [13], a filestore is a complex system that presents interesting challenges for specification and verification. For example, how do we ensure reliability in the presence of concurrent accesses or how do we deal with accidental failures that may occur during the execution.

* This work was part of the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 25–44, 2008.
© Springer-Verlag Berlin Heidelberg 2008

The file system is chosen as a case study for our experiment which is carried out by using Event-B [3] and the Rodin platform [8, 2] for specification, refinement and proof.

We make strong use of refinement to introduce gradually features to the formal model. We see the contribution as twofold. Firstly our work provides evidence of the applicability of the Event-B language, the refinement approach and of the Rodin tool. Secondly our experiment provides guidance on effective modelling and proof styles that may be of benefit to others working on formal development of similar systems.

Our specification of this system is focused on a tree structure and basic functionalities affecting the tree structure: create, delete, copy and move objects that can be files or directories. In this specification, we start with an abstract level accompanied with careful formulation of invariants, and then follow this by refinements in which more details are added.

In the abstraction, we introduce the two main properties of a tree structure: (i) there are no loops in a tree structure and (ii) every node in a tree is reachable from the root. For the no-loop property, instead of using transitive closure which is generally used for specification of absence of loops, we employ a no-loop property proposed in [3] to formulate a simpler invariant satisfying this property. Employing this property, which is less complicated than transitive closure, makes the model easier to prove. For the second property, reachability, instead of introducing another invariant, we introduce a machine theorem – which is derived from existing invariants – that can be proved to show that the property is satisfied.

In the first refinement, files and directories are introduced (in the previous abstraction, both files and directories are treated in a similar way as objects). Therefore, in this level, some additional invariants are added to the model. For instance, files and directories are distinct and each object's parent must be a directory.

In addition, other required properties, i.e., a content of file and access permissions, are introduced accompanied with related events concerning these properties in the second and the third refinement, respectively. Some constraints are covered in these two refinements, such as, each file has a content, each object has an owner and its permissions, accessing to each object is dependent on the permissions allowed, etc.

In total 162 proof obligations were automatically generated by the Rodin platform. 78% of them are proved automatically while others are discharged by using the interactive prover. Based on interactive proof, we introduced some proved theorems that can be reused for discharging several similar proof obligations. This makes interactive proof easier. Consequently, the time required was also reduced. An archive of our development in the Rodin tool may be downloaded [1]. This can be imported by anyone with an installation of the tool which is freely available [2].

[1] http://deploy-eprints.ecs.soton.ac.uk/22/
[2] www.event-b.org

This paper will begin with providing a short description of Event-B and the Rodin platform. Secondly, an informal description of a tree-structured file system and its constraints are given in Section 3. Thirdly, an Event-B specification of the file system which is divided into four levels (an abstraction and three levels of refinements) will be outlined in Section 4, 5, 6, 7 and 8. Fourthly, in Section 9, proof statistics will be figured. Finally, comparison with related work and conclusion will be given in Section 10 and 11 respectively.

2 Event-B and the Rodin Platform

Event-B [3] is an extension of the B-method [1] for specifying and reasoning about complex systems including concurrent and reactive systems. An Event-B model is described in terms of contexts and machines, see Fig. 1.

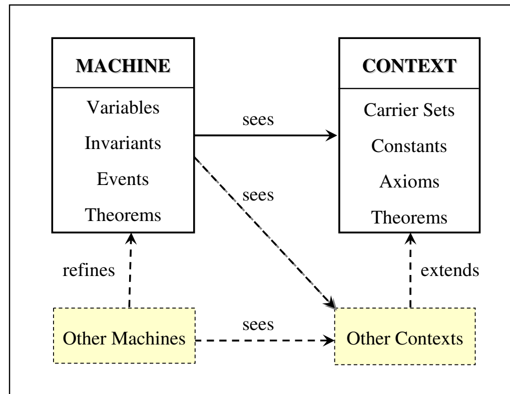


Fig. 1. Relationship between machines and contexts

Contexts [4, 5] contain the static parts of a model. Each context may consist of carrier sets and constants as well as axioms which are used to describe the properties of those sets and constants. Contexts may contain theorems for which it must be proved that they follow from the preceding axioms and theorems. Moreover, contexts can be extended by other contexts and seen by more than one machine. Additionally, a context may be indirectly seen by machines. Namely, a context C can be seen by a machine M indirectly if the machine M explicitly sees another context which is an extension of the context C.

Machines [4, 5] contain the dynamic parts of an Event-B model. This part is used to provide behavioural properties of the model. A machine is made of a state, which is defined by means of variables, invariants, events and theorems shown in Fig. 1. The theorems of a machine must be shown to follow from the context and the invariants of that machine. In addition, machines can be refined by other machines, but each machine can refine at most one machine.

Variables, like constants, correspond to mathematical objects: sets, binary relations, functions, numbers, etc. They are constrained by invariants $I(v)$ where v are the variables of the machine. Invariants are supposed to hold whenever variable values change. But this must be proved through the discharge of proof obligations [4].

A machine contains a number of atomic events which show the way that the machine may evolve. Each event is composed of three elements: an event name, guard(s) and action(s). The guard is the necessary condition for the event. The actions determine the way in which the state variables are going to evolve when performing the event [4].

An event is guarded and atomic and may be performed only when its guard holds. This means that when the guards of several events hold at the same time, then only one of them may be performed at that time. The event is non-deterministically chosen to be performed. Generally, an event, named *Evt*, is presented in one of three possible forms shown in Fig. 2. Where $S(v)$ are generalized substitutions of variable v, $G(v)$ represents a guard of event *Evt*, and t is a local variable [4].

Evt $\widehat{=}$ begin *S(v)* end
Evt $\widehat{=}$ when *G(v)* then *S(v)* end
Evt $\widehat{=}$ any *t* where *G(t,v)* then *S(t,v)* end

Fig. 2. Three posible forms of an event

The Rodin platform [8, 2] is an open and extensible tool for Event-B specification and verification. It contains a database of modelling elements used for constructing system models such as variables, invariants and events. It is accompanied by various useful plug-ins such as a proof-obligation generator, provers, model-checkers, UML transformers, etc [6].

3 An Informal Description of a Tree-Structured File System and Constraints

A tree-structured file system can be described in terms of a collection of objects representing files and directories and a set of operations that may be performed on these objects. The objects are structured as a tree. The tree has only one root directory that cannot be deleted, copied or moved. Each object except the root has only one parent which is a directory. Four operations affecting the tree structure are discussed below.

Create. Create an object in an existing directory. The object can be either a file or a directory.

Copy. Copy an existing object from one place to another place. The destination must exist and must not be a descendant of the object being copied or the

object itself. If the object being copied is a directory, all objects belong to that directory must also be copied to the new location and the copy must have the same structure as the original.

Move. Move an existing object in the tree structure from one place to another place. The destination must exist and must not be a descendant of the object being moved or the object itself.

Delete. Delete an existing object in the file system. In case of deleting a directory, all its descendants must also be removed.

4 Abstract Model

In this abstraction, we begin with an abstract model of a tree-structured file system focusing on tree properties and operations affecting the tree structure. However, files and directories are not distinguished in this level. Instead they are postponed to next refinement given in Section 6. Thus, in this level, both of them are treated in the same way as objects which are nodes of the tree structure. Below is a list of requirements in this level.

Req1.1: The tree has a root node.
Req1.2: All objects except the root node must have a parent.
Req1.3: There are no loops in the tree.
Req1.4: Every node in the tree is reachable from the root node.

Machine variables, invariants formulated to satisfy those required properties mentioned above, and initialised values of each variable are given in Fig. 3. These variables, invariants and initialisation are discussed below.

Variables $objects$, $parent$
Invariants
$inv1.1 : objects \subseteq OBJECT$
$inv1.2 : root \in objects$
$inv1.3 : parent \in objects \setminus \{root\} \rightarrow objects$
$inv1.4 : \forall s \cdot (s \subseteq parent^{-1}[s] \Rightarrow s = \varnothing)$
Initialisation
$objects := \{root\}$
$parent := \varnothing$

Fig. 3. Machine variables, invariants and initialisation of an abstract model

In a context seen by this abstract machine, $OBJECT$ is defined as a carrier set and $root$ is an $OBJECT$ constant (see Fig. 5). Considering Fig. 3, there are two state variables introduced in the machine: (i) $objects$, a set of existing

objects in the file system ($inv1.1$); and (ii) *parent*, a total function mapped from all objects except *root* to their parent which is an object. In this abstraction, *objects* and *parent* are initialised to a set consisting of *root* and the empty set respectively. Invariant $inv1.3$ states that all objects except *root* must have a parent. This invariant satisfies $Req1.2$. Invariant $inv1.4$ is introduced to ensure that there are no loops in the tree structure (satisfying $Req1.3$). This invariant is formulated by using the no-loop property proposed by Abrial in [3]. The reason we choose this formulation, instead of transitive closure which is generally used to specify tree properties – such as a specification of visual file system in [12] – is to make the model simpler and easier to prove.

Considering $inv1.4$, $parent^{-1}[s]$ gives the direct descendants of all elements of set s. For $s \subseteq objects, s \subseteq parent^{-1}[s]$ means that s contains a loop in the parent relationship. Hence, this invariant states that the only such set that can exist is the empty set and thus the parent structure cannot have loops. If we were to use transitive closure, we would need to add the property $inv1.4b$ given in Fig. 4 to the machine invariants.

$$inv1.4b : tcl(parent) \cap id(OBJECT) = \varnothing$$

Fig. 4. No-loop property

Here *tcl* which is mentioned in Invariant $inv1.4b$ is a transitive closure. In a context shown in Fig. 5, *tcl* is defined as a total function mapped from $OBJECT \leftrightarrow OBJECT$ to $OBJECT \leftrightarrow OBJECT$. Giving $r \in OBJECT \leftrightarrow OBJECT$, the transitive closure of r is equal to $r \cup r; tcl(r)$ ($thm1$ of Fig. 5).

The *parent* variable is updated by several of the events. If we were to use $inv1.4b$ instead of $inv1.4$, the *Copy* event, for example, would give rise to a proof obligation with $inv1.4b$ as a hypothesis and the following goal:

$$tcl(parent \cup replica \cup \{nobj \mapsto to\}) \cap id(OBJECT) = \varnothing$$

Proof of this PO would not be easy since distribution of *tcl* through union and other set operations is not straightforward. We avoid such difficulty proofs by using formulation $inv1.4$ instead. Significantly, we can prove that the formulation in $inv1.4b$ follows from the formulation in $inv1.4$. This is given by Theorem $thm3$ shown in Fig. 5. This theorem has been proved using the interactive prover of Rodin. The strategy we follow in proving this theorem is to use proof by contradiction.

In order to satisfy requirement $Req1.4$, instead of introducing another invariant, we present other machine theorems (given in Fig. 6) which are derived from existing invariants and guarantee that the property is satisfied. Considering Theorem $mth3$, since $(tcl(parent))^{-1}[\{root\}]$ returns all objects reachable from *root*, this theorem shows that all objects except *root* are reachable from *root*. Other machine theorems, $mth1$ and $mth2$, are used in the proof of $mth3$. Theorem $mth4$ is introduced to satisfy the no-loop property.

Sets
$OBJECT$
Constants
$root, tcl, objrel, objfn$
Axioms
$axm1 : root \in OBJECT$
$axm2 : objrel = OBJECT \leftrightarrow OBJECT$
$axm3 : objfn = OBJECT \setminus \{root\} \rightarrow OBJECT$
$axm4 : tcl \in objrel \rightarrow objrel$
$axm5 : \forall r \cdot (r \in objrel \;\; \Rightarrow \;\; r \subseteq tcl(r))$
$axm6 : \forall r \cdot (r \in objrel \;\; \Rightarrow \;\; r; tcl(r) \subseteq tcl(r))$
$axm7 : \forall r, t \cdot (r \in objrel \; \wedge \; r \subseteq t \; \wedge \; r; t \subseteq t \;\; \Rightarrow \;\; tcl(r) \subseteq t)$
Theorems
$thm1 : \forall r \cdot (r \in objrel \;\; \Rightarrow \;\; tcl(r) = r \cup (r; tcl(r)))$
$thm2 : tcl(\varnothing) = \varnothing$
$thm3 : \forall t \cdot (t \in objfn \; \wedge \; (\forall s \cdot s \subseteq (t^{-1})[s] \;\; \Rightarrow \;\; s = \varnothing)$
$\Rightarrow \;\; tcl(t) \cap id(OBJECT) = \varnothing)$

Fig. 5. Definition of transitive closure (tcl) and no-loop theorem ($thm3$)

Theorems
$mth1 : \forall T \cdot (root \in T \; \wedge \; parent^{-1}[T] \subseteq T \;\; \Rightarrow \;\; objects \subseteq T)$
$mth2 : objects \subseteq \{root\} \cup (tcl(parent))^{-1}[\{root\}]$
$mth3 : objects \setminus \{root\} \subseteq (tcl(parent))^{-1}[\{root\}]$
$mth4 : tcl(parent) \cap id(OBJECT) = \varnothing$

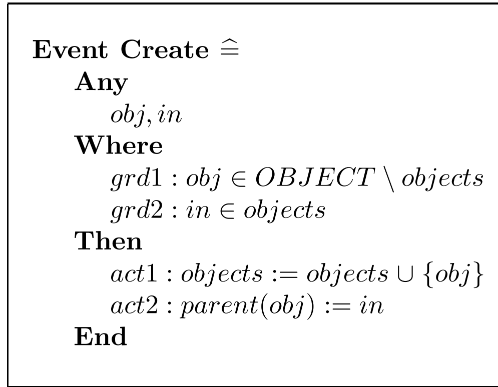
Fig. 6. Machine theorems satisfying reachability and no-loop properties

5 Events

In this section, we outline four abstract events including *Create*, *Move*, *Copy* and *Delete*.

Create event. Create an object in an existing location (see Fig. 7). In the figure, *obj* is an object being created and *in* is its parent. Here *obj* must be an OBJECT that is not already in the set *objects* (see *grd*1); and *in* must exist (see *grd*2). The object *obj* will be added to the set *objects* by *act*1; and *in* will be assigned to be the *obj*'s parent by *act*2.

Move event. This event is aimed at moving an existing object except *root* from one place to another place. Considering Fig. 8, *a* is an object being moved from node *r* to node *c*. Node *c* will become a new parent of *a*. In Fig. 9, an



Event **Create** $\widehat{=}$
  Any
    obj, in
  Where
    $grd1 : obj \in OBJECT \setminus objects$
    $grd2 : in \in objects$
  Then
    $act1 : objects := objects \cup \{obj\}$
    $act2 : parent(obj) := in$
  End

Fig. 7. A specification of Event *Create*

existing object named *obj* is moved to a new location named *to*. Parameter *des* is the set of all descendants of *obj* which is equal to $(tcl(parent))^{-1}[\{obj\}]$. In this case, the destination, *to*, must exist and not be *obj* or a descendant of *obj* (these constraints are specified as $grd2$ and $grd5$). These guards are necessary to guarantee that the move does not introduce a loop or unreachable objects. The *parent* function is updated so that *obj* has *to* as its parent.

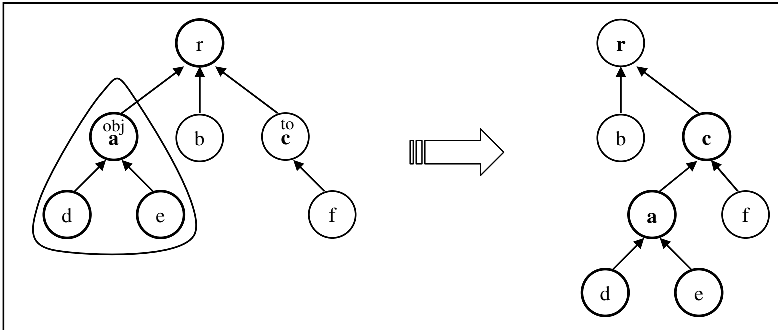


Fig. 8. Diagram of moving a subtree rooted at a from r to c

Copy event. In order to understand more about the copy event, we will describe this event by using Fig. 10. From the figure, the left-hand side is a tree before copying and the right-hand side is the result. Here r is a root node, a is an object being copied (d and e, its descendants, will be copied as well) from node r to node c. The arrows represent the function *parent* and the dashed lines represent a correspondence function which is a bijection from the set of all objects being copied to the set of new objects (a', d', and e') which is a copy of that set. The correspondence bijection is used to maintain the structure of directory a in the copy.

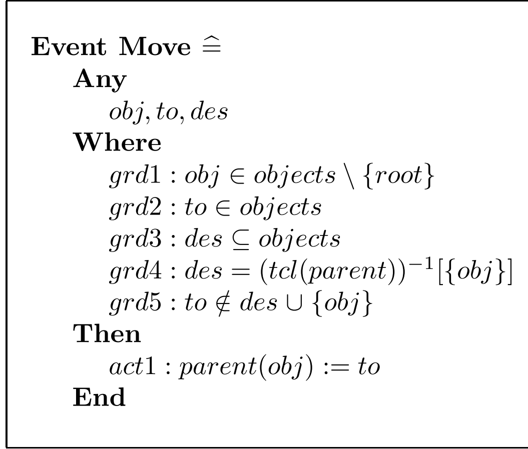


Fig. 9. A specification of Event *Move*

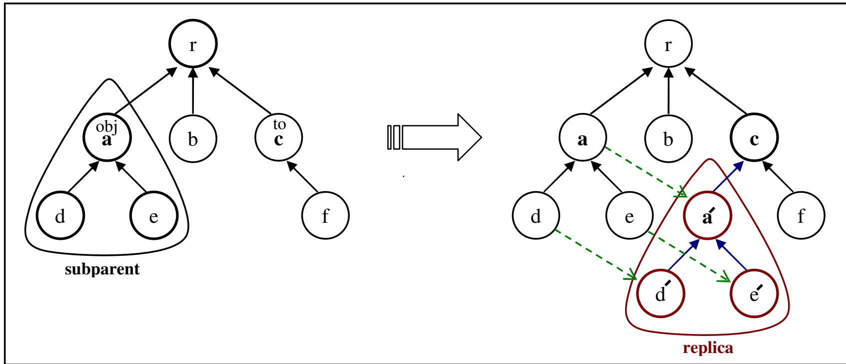


Fig. 10. A diagram of copying a subtree (*subparent*) rooted at a from r to c

Considering Event *Copy* given in Fig. 11, *obj* (the object being copied) and *to* (the destination) behave like external parameters provided by users or application programs, while the rest are local parameters used for computation. However, there is no distinction between external parameters and local parameters in Event-B. In this event, *des* is the set of all descendants of the object *obj* which is equal to $(tcl(parent))^{-1}[\{obj\}]$; *objs* is the set of all objects being copied; *nobjs* is the set of new objects corresponding to the set *objs*; *corres* is the correspondence bijection. With reference to Fig. 10, *subparent* represents the subtree rooted at a which is being copied. In this event, *subparent* is equal to $des \lhd parent$ which is a restriction of the parent function to *des* (e.g., $d \mapsto a$ and $e \mapsto a$ in Fig. 10). Finally, *replica* is a copy of *subparent* which is equal to $corres^{-1}; subparent; corres$ (e.g., $d' \mapsto a'$ and $e' \mapsto a'$ in Fig. 10).

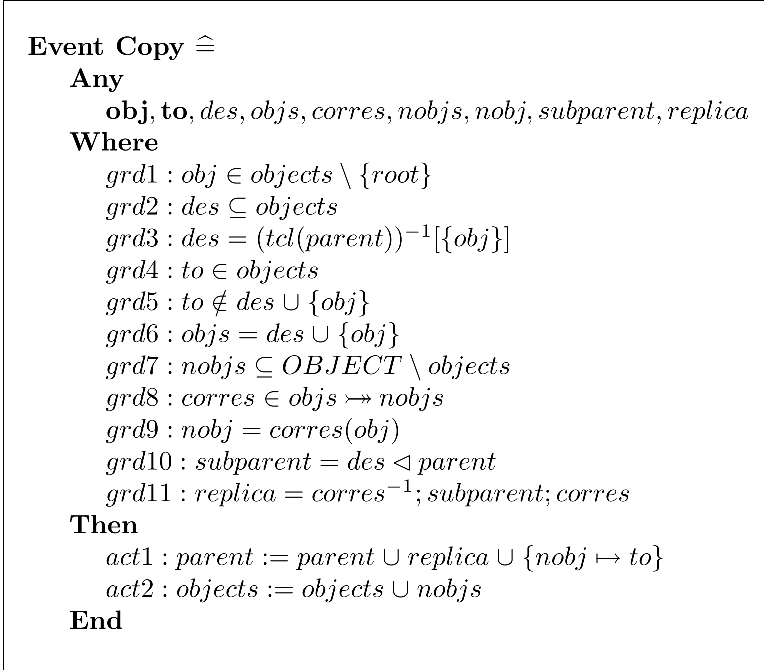


Fig. 11. A specification of Event *Copy*

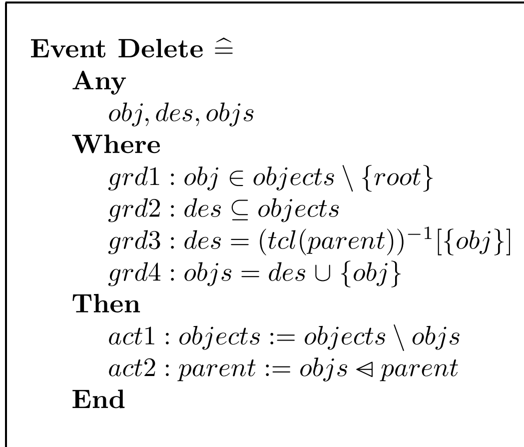


Fig. 12. A specification of Event *Delete*

At this point, the reason we introduce a number of additional local parameters is to make models easier to read and prove. For example, without introducing *des*, *subparent* and *replica*, *act*1 can be replaced by

$$parent := parent \cup corres^{-1}; (tcl(parent))^{-1}[\{obj\}] \lhd parent; corres \\ \cup \{nobj \mapsto to\}$$

but we can see that the action becomes more difficult to read.

Additionally, there are two main constraints in this event. Firstly, the object being copied, obj, must exist and must not be the $root$. This is satisfied by $grd1$. Secondly, the destination, to, must exist and must not be the object being copied or its descendant (satisfied by $grd5$). Guard $grd5$ plays an important role to ensure that loops are not produced by this event.

Delete event. This event is given in Fig. 12. In the figure, obj is an object being deleted; des is a set of all obj's descendants. Here $grd1$ states that obj must be an existing object except $root$. The object being deleted and all its descendants, $objs$, will be removed from $objects$ by $act1$ and all related parent-entries also be removed by $act2$.

6 First Refinement: Files and Directories

In this refinement, objects are classified as files or directories. There are two machine variables introduced in this level, namely, $files$ (a set of existing files) which is initialised to the empty set and $directories$ (a set of existing directories) which is initialised to a set of $root$. The variables $files$ and $directories$ are used to partition the variable $objects$. Additionally, the $Create$ event of the abstraction is refined into events $crtfile$ (create file) and $mkdir$ (make directory). Additional requirements for this level are given below.

Req2.1: Set of objects is partitioned into files and directories.
Req2.2: Root node is a directory.
Req2.3: The parent of each object must be a directory.

Variables
$files,\ directories,\ parent$
Invariants
$inv2.1 : files \subseteq objects$
$inv2.2 : directories \subseteq objects$
$inv2.3 : files \cap directories = \varnothing$
$inv2.4 : objects = files \cup directories$
$inv2.5 : root \in directories$
$inv2.6 : ran(parent) \subseteq directories$
Initialisation
$files := \varnothing$
$directories := \{root\}$
$parent := \varnothing$

Fig. 13. Machine variables, invariants and initialisation of the first refinement

Fig. 13 shows a list of machine variables, invariants formulated to satisfy above requirements and initialised values of each variable. Considering the gluing invariant $inv2.4$, the abstract variable $objects$ is entirely defined in terms of $files$ and $directories$. As a result, it can be substituted by $files \cup directories$ and is no longer used in this level.

In this refinement, we choose two events (*Create-file* and *Copy*) to illustrate a concrete model of this level.

Create-file event. This event (named $crtfile$), given in Fig. 14, refines $Create$ of the previous abstraction. Additional details introduced in this refinement: (i) $grd2$, in must be a directory; and (ii) $act1$, the object must be added to the set $files$ directly, instead of the set $objects$ in the previous abstraction.

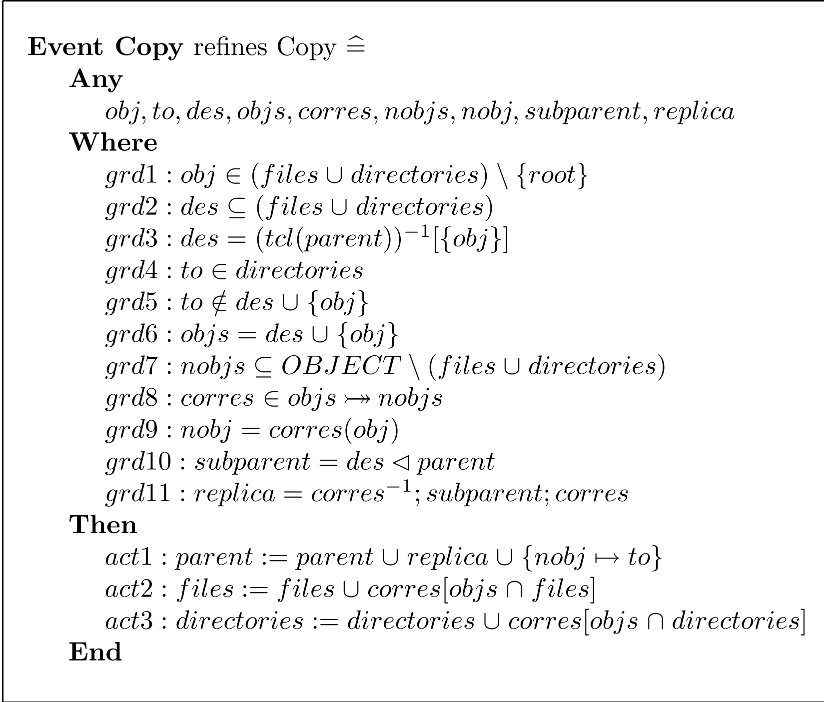
Event crtfile refines Create $\widehat{=}$
  Any
    obj, in
  Where
    $grd1 : obj \in OBJECT \setminus (files \cup directories)$
    $grd2 : in \in directories$
  Then
    $act1 : files := files \cup \{obj\}$
    $act2 : parent(obj) := in$
  End

Fig. 14. A specification of Create-file event

A refinement of Event ***Copy*****.** In this refinement, see Fig. 15, additional details introduced in this event are: (i) $grd4$, the destination, to, must be a directory; (ii) $act2$, all correspondents of $objs$ which are files must be added to the set $files$; and (iii) $act3$, all correspondents of $objs$ which are directories must be added to the set $directories$ as well. These two actions refine Action $act2$ of the previous abstraction.

7 Second Refinement: File Content

In this refinement, file contents and other requirements related to the contents are introduced accompanied with four events: *open* (open an existing file), *read* (read the whole content of a file from the storage into memory buffer), *write* (write the content of a file on the buffer back to the storage) and *close* (close an opened file). Some constraints are covered in this level – such as each file has some content; each file must be opened before reading or writing; and a buffer of each opened file will be assigned once the file is opened and released when the file is closed. Machine variables introduced in this refinement are listed and discussed below.



Event Copy refines Copy $\widehat{=}$

Any

$obj, to, des, objs, corres, nobjs, nobj, subparent, replica$

Where

$grd1 : obj \in (files \cup directories) \setminus \{root\}$

$grd2 : des \subseteq (files \cup directories)$

$grd3 : des = (tcl(parent))^{-1}[\{obj\}]$

$grd4 : to \in directories$

$grd5 : to \notin des \cup \{obj\}$

$grd6 : objs = des \cup \{obj\}$

$grd7 : nobjs \subseteq OBJECT \setminus (files \cup directories)$

$grd8 : corres \in objs \rightarrowtail\!\!\!\!\twoheadrightarrow nobjs$

$grd9 : nobj = corres(obj)$

$grd10 : subparent = des \lhd parent$

$grd11 : replica = corres^{-1}; subparent; corres$

Then

$act1 : parent := parent \cup replica \cup \{nobj \mapsto to\}$

$act2 : files := files \cup corres[objs \cap files]$

$act3 : directories := directories \cup corres[objs \cap directories]$

End

Fig. 15. A refinement of Event *Copy*

$$fcontent \in files \rightarrow CONTENT$$

$$opened_files \subseteq files$$

$$fbuffer \in opened_files \rightarrow CONTENT$$

In this refinement, the content of files, $fcontent$, is defined as a total function mapped from each file to a content. Variable $opened_files$ is set of files which are opened. The buffer of each opened file, $fbuffer$, is specified as a total function mapped from each opened file to a content. The content is an array of data items (BYTEs). In a context seen by this refined machine, the content is defined as a constant named $CONTENT$; and $BYTE$ is defined as a carrier set.

$$CONTENT = \mathbb{N} \leftrightarrow BYTE$$

Fig. 16 given below is an example of event *read*. This event is aimed at reading the whole content of a file named f from a storage into its buffer. Guard $grd1$ states that the file f must be an opened file.

8 Third Refinement: Permissions

In this level, requirements related to access permissions are introduced. For example, each object has an owner, a group-owner and a list of permissions. Access

Event read $\widehat{=}$
  Any
    f
  Where
    $grd1 : f \in opened_files$
  Then
    $act1 : fbuffer(f) := fcontent(f)$
  End

Fig. 16. A specification of Event *read*

to each object depends on its permissions. Additionally, users and groups are specified in this level as well. For instance, each user can be a member of one or more groups but at most one primary group is assigned, etc.

Considering Fig. 17, a number of machine variables are introduced in this refinement: *users*, a set of existing users; *groups*, a set of existing groups; *user_pgrp*, a primary group of each user; *user_grps*, user's groups; *obj_owner*, an owner of each object; and *obj_perms*, permissions of each object. Invariant $inv4.5$ states that a primary group of each user must be a group in which the user be a member. In a context seen by this machine, GROUP, USER and PERMISSION (a set of permission types) are defined as carrier sets.

Variables
  ...
  $users,\ groups,\ user_pgrp,\ user_grps,\ obj_owner,\ obj_grp,\ obj_perms$
Invariants
  $inv4.1 : users \subseteq USER$
  $inv4.2 : groups \subseteq GROUP$
  $inv4.3 : user_pgrp \in users \rightarrow groups$
  $inv4.4 : user_grps \in users \leftrightarrow groups$
  $inv4.5 : \forall u \cdot u \in users \Rightarrow user_pgrp(u) \in user_grps[\{u\}]$
  $inv4.6 : obj_owner \in (files \cup directories) \rightarrow users$
  $inv4.7 : obj_grp \in (files \cup directories) \rightarrow groups$
  $inv4.8 : obj_perms \in (files \cup directories) \leftrightarrow PERMISSION$

Fig. 17. Additional machine variables and invariants of the third refinement

Fig. 18 is an example of Event *read*, which refines the *read* event of the previous abstraction. In this event, guards $grd2$ and $grd3$ state that user *usr* who issues this read request must exist and has a read permission on f.

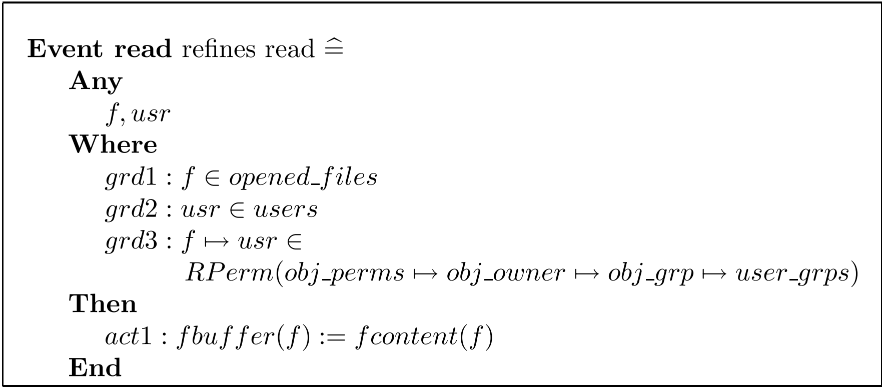


Fig. 18. A refinement of Event *read*

$RPerm$, which is mentioned in the event *read*, encodes the rules that determine whether a user has read permission for a file. It is defined in a context seen by this machine. Part of this context which is related to $RPerm$ is shown in Fig. 19. In the figure, p represents a permission relation; s is an owner function; g is an object-group function; m is a user-group relation; su, a super user (who has the right to manage every thing), is defined as a USER constant; and rbo (owner-read), rbg (group-read) and rbw (world-read) are permission types. This function states that a user u has a permission to read an object o only if at least one of these criteria is satisfied: (i) the user is the owner and has the owner-read permission; (ii) the user is a member of the group to which the object belongs and has the group-read permission; (iii) the world-read permission is assigned to the object; and (iv) the user is the super user. Other permission definitions (i.e., write and execute permission functions) which are not mentioned here are also specified in the same way.

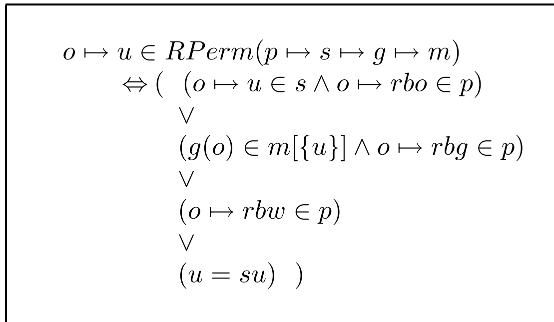


Fig. 19. A definition of read permission function

9 Proofs

The proof statistics, given in Fig. 20, show that 162 proof obligations were generated by the Rodin platform. 127 proof obligations (or 78%) were proved automatically while others were discharged by interactive proof. In the figure, MCH0 represents an abstract model; MCH1, MCH2 and MCH3 represent the first, second and third refinements of the abstract model. CTX0 up to CTX3 represent corresponding contexts which are seen by those machines.

Machines/Contexts	Total POs	Automatic	Interactive
CTX0	10	8	2
CTX1	7	3	4
CTX2	0	0	0
CTX3	3	3	0
MCH0	35	22	13
MCH1	50	42	8
MCH2	17	15	2
MCH3	40	34	6
Overall	162	127 (78%)	35 (22%)

Fig. 20. Proof statistics

In order to make proof easier and reduce the time required, we introduced proved theorems that could be reused for discharging some similar proof obligations. The example given in Fig. 21 is a theorem introduced in a context seen by the abstract machine. This theorem was used to prove that the no-loop property held for Event *Copy*. To prove this event preserves the no-loop property ($inv1.4$), we provided: $f = parent$, $g = replica$, $r = root$, $u = nobj$, $x = to$, $M = objects$ and $N = nobjs$. However, the theorem could be reused for Event *Create* and *Move* events as well. For example, in the case of Event *Create*, g was assigned to be the empty set, $u = obj$ and $N = \{obj\}$.

Proof of Theorem $thm4$ was one of the most complex of the interactive proofs. We outline the steps involved. Proving $thm4$, we have the goal $G1$:

$$C = \varnothing \tag{G1}$$

with the hypothesis $H1$:

$$C \subseteq (f \cup g \cup \{u \mapsto x\})^{-1}[C] \tag{H1}$$

$H1$ is rewritten to $H2$:

$$C \subseteq f^{-1}[C] \cup g^{-1}[C] \cup \{x \mapsto u\}[C] \tag{H2}$$

Now we cannot prove that $C \subseteq f^{-1}[C]$, but we can use $H2$ and other antecedents of $thm4$ to prove

$$C \cap M \subseteq f^{-1}[C \cap M] \tag{H3}$$

$$
\begin{array}{l}
thm4 : \forall f, g, r, u, x, M, N \cdot \\
\quad M \subseteq OBJECT \land\ N \subseteq OBJECT \land\ M \cap N = \varnothing \\
\quad \land\ r \in M \land f \in M \setminus \{r\} \rightarrow M \\
\quad \land\ u \in N \land g \in N \setminus \{u\} \rightarrow N \\
\quad \land\ x \in M \\
\quad \land\ (\forall A \cdot A \subseteq f^{-1}[A] \Rightarrow A = \varnothing) \\
\quad \land\ (\forall B \cdot B \subseteq g^{-1}[B] \Rightarrow B = \varnothing) \\
\quad \land\ f \cup g \cup \{u \mapsto x\} \in (M \cup N) \setminus \{r\} \rightarrow M \cup N \\
\Rightarrow \\
\quad (\forall C \cdot C \subseteq (f \cup g \cup \{u \mapsto x\})^{-1}[C] \Rightarrow C = \varnothing)
\end{array}
$$

Fig. 21. A theorem used for discharging no-loop proof obligation

From $H3$ and the antecedent of $thm4$ we can prove

$$C \cap M = \varnothing \tag{H4}$$

Similarly, we can prove that

$$C \cap N = \varnothing \tag{H5}$$

We observe that C can be partitioned by M and N. Thus, using $H2$ we can prove

$$C = (C \cap M) \cup (C \cap N) \tag{H6}$$

Finally, $G1$ is proved using $H4$, $H5$, and $H6$.

10 Comparison with Related Work

A number of formalisations of file systems have been developed by other researchers. Most of them are focused on file contents, and read and write operations. There is some work that deals with the structure of file systems. A specification of a visual file system in Z by Hughes [12] is focused on a tree structure and operations affecting the tree structure, but file content and a manipulation of file content were not specified. In this specification, transitive closure was chosen to specify main property of a tree structure, e.g. reachability. However, the no-loop property was not mentioned in this specification. In addition, this specification had no refinement and no proof. Another related work by Morgan and Sufrin presented in [11] is a specification of a Unix filing system in Z. In this specification, instead of using a tree structure, the location of each object is formulated as a sequence of directory names, which is the path of each file. This work is concentrated on file contents and naming operations used for manipulating these rather than structure manipulation operations such as directory copy and move. Based on the specification of Morgan and Sufrin, Freitas,

Fu and Woodcook [10] have developed a verified model of the POSIX filestore accompanied with a representation and proof using the Z/Eves proof system.

Since the filestore challenge was proposed by Joshi and Holzmann [13] in 2005, other researchers have addressed this challenge. For example, Butterfield and Woodcook [7] have developed an abstract specification in Z of the ONFi specification [16]. In addition, Ferreira et al. [9] have developed and verified a specification of the Intel Flash File System Core [17] in VDM. Alloy and HOL were used as tools for model checking and theorem proving. Another work contributed by Kung and Jackson [14] is a formal specification and analysis of a flash-based file system in Alloy. This work focused on basic operations of a filesystem and features covering wear-levelling and fault tolerance.

11 Lessons and Conclusion

In this paper, we have presented a verified model of a tree-structured file system focusing on the tree structure and the basic operations affecting the tree structure. Our aims are constructing a model with clear and accurate formulation of the system properties and discharge of all proof obligations. To satisfy these, careful selection of invariants and machine theorems was important and eased the proof effort. For example, for the high-level requirements on the data structure, we introduced two tree properties: (i) no-loop and (ii) reachability. Both these properties are naturally expressed using transitive closure. However, we identified simpler but sufficient formulations ($inv1.3$, $inv1.4$) and exposed these as invariants. Proving that all events preserved these invariants was not too difficult since they did not involve transitive closure. The transitive closure formulations were expressed as machine theorems and we proved that these followed from the machine invariants. We did not need to prove that the theorems are preserved by all machine events which simplified the proof effort considerably.

Our experience of using the Rodin tool was very positive[1]. The supported language was sufficiently expressive and all proof obligations could be discharged. We achieved a good degree of automatic proof. All interactive proofs involved a small number of steps and were straightforward to achieve.

Based on this experience, we have found that general theorems should be specified in a context such as Theorem $thm4$ in Fig. 21. They can be seen and used by more than one machine, and can be extended by other contexts. Specific theorems which are derived from machine variables and invariants should be specified in machines (such as machine theorems given in Fig. 6). These machine theorems can be used to help discharge proof obligations as well. Introducing additional theorems that can be reused for discharging similar proof obligations makes automatic and interactive proof easier and can reduce the time required for proofs. In addition, instead of introducing new machine invariants to satisfy system properties, providing machine theorems and proving that those properties

[1] Caveat: Abrial and Butler are developers of the Rodin tool so are not objective evaluators.

are satisfied is another mechanism used to specify system models. This mechanism can reduce the number of proof obligations and makes models simpler and easier to prove.

Additionally, it can be seen in an example given in Section 5, providing additional parameters in each event is useful sometimes. Although more guards are needed, it could make models more readable and easier to manage in both specifying and proof.

Refinement can be used to introduce other requirements that may be postponed or missed from the previous steps and later be covered in the refinement steps. Refinement allows us to factor out some of the modelling and proof complexities. In this development we chose to focus on the tree structure manipulation in the abstract model and postpone other details to later refinements - for example, we do not distinguish files and directories at the abstract level. This made the proof obligations and invariants for the tree structure easier to formulate than if we had tried to model everything in one level. Note that we regards the full chain of refinements as constituting *the specification*, not just the most abstract level.

Finally, it can be stated that this example allowed us to define a kind of modelling methodology – finding the right mathematical concepts, finding useful general theorems – which could be exported in many different complex modelling projects which require a manipulation of the tree structure.

References

1. Abrial, J.-R.: The B Book. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: A system development process with Event-B and the Rodin platform. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 1–3. Springer, Heidelberg (2007)
3. Abrial, J.-R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2008)
4. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260. Springer, Heidelberg (2006)
5. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. Fundamentae Infomatica, 1001–1026 (2006)
6. Butler, M.: Rodin deliverable D31: Plublic versions of plug-in tools. Technical report, University of Southampton, UK (2007)
7. Butterfield, A., Woodcock, J.: Formalising flash memory: First steps. In: 12th ICECCS 2007, pp. 251–260. IEEE Computer Society Press, USA (2007)
8. Coleman, J., Jones, C., Oliver, I., Romanovsky, A., Troubitsyna, E.: RODIN (Rigorous open Development Environment for Complex Systems). In: 5th European Dependable Computing Conference: EDCC-5 supplementary, Budapest, pp. 23–26 (2005)
9. Ferreira, M.A., Silva, S.S., Oliveira, J.N.: Verifying Intel flash file system core specification. Technical report, University of Minho (2008)
10. Freitas, L., Fu, Z., Woodcock, J.: POSIX file store in Z/Eves: an experiment in the verified software repository. In: 12th ICECCS, pp. 3-14 (2007)

11. Hayes, I.: Specification Case Studies. Prentice Hall International, UK (1992)
12. Hughes, J.: Specifying a visual file system in Z. Technical report, Department of Computing Science, University of Glasgow (1989)
13. Joshi, R., Holzmann, G.J.: A mini challenge: Build a verifiable filesystem. In: Verified Software: Theories, Tools, Experiments (2005)
14. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. 1st Conference on ASM, B, and Z (ABZ 2008). London, UK (to appear) (September 2008)
15. Métayer, C., Abrial, J.-R., Voisin, L.: Rodin deliverable 3.2. Event-B language. Technical report, University of Newcastle upon Tyne, UK (2005)
16. Cemicondutor, H., et al.: Open NAND Flash Interface Specification. Technical report Revision 1.0, ONFI (December 2006), http://www.onfi.org
17. Intel Flash File System Core Reference Guide, version 1. Technical report 304436001, Intel Coorporation (October 2004)