Modular, Crash-Safe Refinement for ASMs with Submachines

Gidon Ernst[a], Jörg Pfähler[a], Gerhard Schellhorn[a], Wolfgang Reif[a]

[a]*Institute for Software & Systems Engineering*
University of Augsburg, Germany

Abstract

In this paper we define a formal refinement theory for a variant of Abstract State Machines (ASMs) with submachines and power cuts. The theory is motivated by the development of a verified flash file system. Different components of the system are modeled as submachines and refined individually. We define a non-atomic semantics that is suitable for considering power cuts in the middle of operations. We prove that refinement is compositional with respect to submachines and crashes. We give a criterion "crash-neutrality" and corresponding proof obligations that are sufficient to reduce non-atomic reasoning to standard pre/post verification in the context of power failures in file systems.

Keywords: Abstract State Machine, Refinement, Compositionality, Flash File Systems, Crash Safety, Power Cuts

1. Introduction

This paper contributes a variant of Abstract State Machines (ASMs, [1, 2]) termed *Data Type Abstract State Machines* and a formally defined instance of refinement theory. They provide strong *modularity* guarantees with respect to submachines that respect information hiding, and are able to handle *crashes* during runs, which are caused by external events such as power cuts.

Motivation for this work is our current effort to construct a verified file system for flash memory. File system verification has been proposed as a challenge by NASA [3] in response to problems with the Mars Rover "Spirit" [4]. At the time of writing, we have solved a large part of this challenge; some background is presented in Sec. 2. For a complete overview, see [5] and our web-presentation [6]. The project is realized with the theorem prover KIV [7].

The verification shows two desired properties of the file system. *Functional correctness* with respect to an abstract POSIX specification ensures that the

[☆]This work is part of the project "Verifikation von Flash-Dateisystemen" (RE828/13-1) sponsored by the Deutsche Forschungsgemeinschaft (DFG).

Email addresses: `ernst@isse.de` (Gidon Ernst), `pfaehler@isse.de` (Jörg Pfähler), `schellhorn@isse.de` (Gerhard Schellhorn), `reif@isse.de` (Wolfgang Reif)

system will function as expected under normal conditions. *Crash-safety* ensures that unexpected *power cuts* or similar events have a well-specified, limited effect only; in practice this means that top-level operations take effect atomically. We formally present our approach that has made the verification feasible: the given notion of refinement is *compositional*, i.e., both standard functional correctness as well as crash-safety can be decomposed into verification conditions for each individual refinement.

Since a power failure can happen at any point in time in the final implementation, we can not just rely on an atomic view of operations. The foundation for the refinement theory is therefore a fine-grained, *non-atomic* semantics (defined in Sec. 3) that exposes sequences of intermediate states of computations. The *atomic* semantics of rules can be reconstructed from the non-atomic view by collapsing the sequence of states into the respective initial and final state. The *crashing* semantics (defined in Sec. 6) executes a prefix of the non-atomic trace, erases non-persistent data (conceptually, the part of the state that is stored in RAM), followed by a recovery operation to restore a consistent state.

The means to achieve a modular development is by using *submachines* to model different components of the system with varying degrees of abstraction. Examples are given in Sec. 2. Taking up ideas from contract refinement as used in Z [8], the machines in this work have operations with preconditions and designated inputs/outputs, and an internal state that is encapsulated.

Non-crashing runs are based on the atomic semantics of rules and induce a standard notion of refinement that can be proved by forward simulation, as formalized in Sec. 4. Refinement of an abstract specification to a concrete implementation guarantees that the latter can be substituted for the former in a given context. We model the context explicitly as an outer ASM, which allows us to prove the main theorem of Sec. 5, namely that this approach is sound in our setting: refinement composes (recursively) with respect to submachines. Runs with power cuts refer to the crashing semantics instead of the atomic one, embedding crash-safety into the previously established refinement theory (see Sec. 6). In particular, the abstract crash specifies what an implementation has to guarantee after a crash. The novel semantic aspect here is the way we abstract the non-atomic view of operations to an atomic one. Within the fine-grained semantics of an operation of the outer machine, a call to an operation of the submachine is just a single step. Provided a simple condition termed "crash-neutrality" holds, this approach will provide the lever to switch from a fine-grained "white-box" analysis of crashes to a much simpler atomic "black-box" view—in a way that is generic and provably compatible with submachine refinement in Sec. 7.

This paper extends the work in [9]. Some of the definitions have been slightly simplified: the handling of input and output of data type ASM rules is simpler, and we now only have one type of intervals, although we extend them twice (with $\perp$ for nontermination of a submachine call in Sec. 5 and with $\lightning$ for crashes in Sec. 6). We also give more details on the proofs. The main contribution here are Sec. 6 and 7, where we extend the theory to consider crashes. The semantic definitions as well as the theorems that yield simple pre/post verification

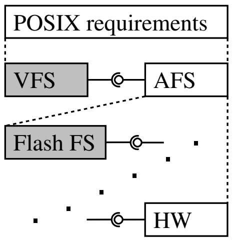

Figure 1: FFS upper layers. The bold dots indicate a series of nested refinements that have been omitted here.

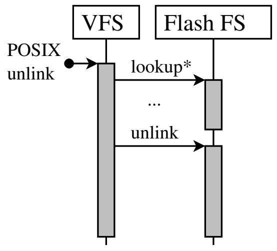

Figure 2: Call sequence in the final composed file system code.

POSIX: **state** *tree*, *fs*, *ofh*

```
posix_unlink(path)
  let  ino = tree[path].ino
  in   tree := tree − path;
       if  ino ∉ files(ofh) ∧ ino ∉ links(tree)
       then fs[ino] := undef
```

Figure 3: POSIX specification of `unlink`.

VFS: **state** *ofh*

```
vfs_unlink(path)
  ino := ROOT_INO;
  while path.length > 1 do {
     afs_lookup(ino, path.head; ino′);
     path := path.tail, ino := ino′
  };
  afs_unlink(ino, path.head);
  if  ino ∉ files(ofh)
  then afs_evict(ino)
```

AFS: **state** *dirs*, *files*

```
afs_unlink(ino, name)
  pre:   dirs[ino] ≠ undef
  dirs[ino].entries[name] := undef
afs_evict(ino)
  pre:   files[ino] ≠ undef
  if links(ino, dirs) = ∅
  then files[ino] := undef
```

Figure 4: VFS/AFS rules (omitting permissions checks and error handling).

conditions for crash-safe refinement are entirely new.

2. Submachines in the Flash File System

In this section, we briefly show the topmost refinement of the refinement hierarchy: Fig. 1 shows an excerpt of the structure of the project, where boxes represent components, connected by refinement (dotted lines). These are formally given by data type ASMs with algebraic states. The grey boxes are the leaves of the hierarchy and constitute the final implementation.

At the toplevel, POSIX [10] specifies the requirements. On this abstract level, the file system (FS) is represented as an acyclic graph consisting of directories (internal nodes) and files (leaves). An example is shown in Fig. 5. Files can be referred to by multiple directories under different names ("hard-links"), consequently, names are attached to edges of the graph. The directory part is a proper tree. The POSIX interface is based on *paths*, which identify files/directories and are looked up stepwise starting from the root.

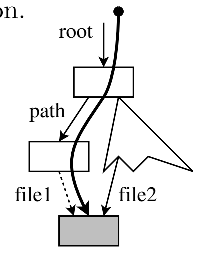

Figure 5: FS graph

Real file system implementations consist of two parts. Generic aspects, i.e., traversing paths and checking access rights are realized by the *Virtual Filesystem Switch* (VFS), similarly to the VFS that is part of the Linux kernel. Concepts specific to a file system implementation are realized by the individual file systems (Flash FS). VFS communicates with them through a well-defined interface. Its main data structure is called *inodes* in Linux. This interface is specified in terms of the *Abstract File System* (AFS), which technically is a *submachine* of VFS, we denoted by VFS(AFS). Interfaces are visualized by the symbol —◎— in Fig. 1. That VFS(AFS) correctly implements the POSIX specification is established by a refinement proof, which is denoted by dotted lines in Fig. 1.

The development continues further down the refinement chain by implementing the AFS specification with flash specific constructs in the model Flash FS. While the *proof* relates VFS(AFS) to POSIX, the final code will be composed of the implementation level models, i.e., the grey models VFS(Flash FS(...)) in Fig. 1. The size of implementation components is much larger than the size of their respective specifications (POSIX: 50 lines, VFS: 500 lines, AFS: 100 lines, Flash FS: 500 lines). This pattern repeats all the way down to the hardware interface HW, forming a *deep* hierarchy: The final system consists of 19 models and 9 refinements in total (see [6, 5]), each with its own formalization of state and invariants. An approach that exploits the compositional structure *and* works for power cuts makes the verification of the whole file system tractable.

As an example, Fig. 3 shows the specification of the POSIX operation `posix_unlink`, which removes one link to a file denoted by *path*. The state of the machine is modeled by an algebraic directory tree *tree*, a map from file identifiers (inode numbers *ino*) to their contents *fs* and a map of open file handles *ofh* with information about access to the file and the current offset. The operation conditionally deletes the file's content from *fs* as well, given that it is not referenced from the *tree* (under a different path) and all file handles to this file have been closed.

The POSIX standard explicitly permits files that are not accessible from the tree any more but are still open. These files are called *orphans* and must be deleted upon recovery. We explain the issue in more depth in Sec. 6.1. Orphans are practically relevant, e.g. when a binary of a running process is overwritten with a new version during a system update, the process still has a handle to the old version of the file and can keep on running.

The realization of the corresponding operation `vfs_unlink` in VFS is shown in Fig. 4 (full models can be found in [11]). Several calls of `afs_lookup` are used to traverse the path, checking that the individual directories exist with suitable access rights. Then, `afs_unlink` is called for the actual removal of the link in the target directory. Operation `afs_unlink` has a *precondition* to characterize valid inputs, which needs to be checked at every call site, in this case that the parent directory *ino* actually exists.

Knowledge of whether the file's content has to be deleted is now split between the two machines (corresponding to the two conjuncts in Fig. 3). The generic aspect of file handles is part of VFS, whereas determining the number of links depends on the actual data representation of the flash file system and lies therefore in the responsibility of AFS as part of an operation called `afs_evict` (which can thus be refined further). For the proof of refinement from POSIX to VFS(AFS), however, the internal state AFS will be exposed; formally, *dirs* is correlated to *tree* and *files* to *fs*, respectively and *ofh* is mapped by identity. The mapping and the proof is described in [12].

Fig. 2 shows the corresponding sequence of operations in the final *composed code* we generate (marked grey in Fig. 1). In this code calls to abstract AFS operations have been replaced by calling the concrete FS code.

3. Syntax and Semantics of Data Type ASM rules

Sec. 3.1 defines the syntax of data type ASM rules. We use only part of the syntax available in [1], and avoid parallel rules which could result in clashes (cf. Sec. 8 for a discussion of the limitations and on possible extensions). The rules are given a non-atomic semantics in Sec. 3.2 that is similar to the one of control state ASMs, however, we never use an explicit control state. We then abstract to an atomic relational view in Sec. 3.3, which serves as the foundation for the weakest-precondition calculus in Sec. 3.4 that we use in mechanized proofs.

3.1. Syntax

We assume the reader is familiar with first-order logic, where, based on a signature $SIG = (F, P)$ with functions $f \in F$ and predicates $p \in P$, terms t, formulas φ and boolean expressions ε (= quantifier-free formulas) can be defined. The semantics $[\![t]\!](s)$ of terms t and the semantics $s \models \varphi$ of formulas φ is defined over a state $s \in S$ consisting of an algebra and a valuation for variables x as usual. We assume the signature is partitioned into four parts: a *static* signature (no updates allowed), an *input* signature (that is only read by rules), an *output* signature (that is only written by rules), and a *controlled* signature that may be read and written by rules.

We use the general convention to underline tuples of elements, i.e., $\underline{a}$ stands for a tuple $a_1, \ldots, a_n$ for some $n \geq 0$. We write $s\{x \mapsto a\}$ for the modified state, where variable x now maps to value a, and $s\{f(\underline{t}) \mapsto a\}$ for the state, where function f has been updated to have value a for arguments $[\![\underline{t}]\!](s)$. A location *loc* is either a variable x or $f(\underline{t})$, so $s\{loc \mapsto a\}$ denotes a generic update. We introduce the abbreviation $s\{loc \mapsto t\} = s\{loc \mapsto [\![t]\!](s)\}$ for terms t, and the generalization $s\{\underline{loc} \mapsto \underline{t}\}$ to a parallel update, when all locations are different. The leading symbol of a location is x and f, respectively. An input (resp. output) location is a location $f(\underline{t})$ where the leading symbol f is in the input (resp. output) signature. We use the following syntax for our rules α, β:

$$\begin{aligned}\alpha ::= \ & \underline{loc} \coloneqq \underline{t} \quad | \quad \alpha;\beta \quad | \quad \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta \quad | \\ & \textbf{while } \varepsilon \textbf{ do } \alpha \quad | \quad \textbf{choose } \underline{x} \textbf{ with } \varphi \textbf{ in } \alpha \textbf{ ifnone } \beta\end{aligned}$$

For parallel updates we require that the leading symbols of $\underline{loc}$ are all distinct and writable, i.e., are local variables or part of the controlled or output signature. We write **skip** for an empty parallel update. Compared to the full generality of ASM rules we have avoided parallel rules/updates that may result in clashes.

The **choose** construct binds local variables $\underline{x}$ to values such that φ is satisfied and executes α. If there is no possible choice (e.g. if $\varphi \equiv \texttt{false}$) then β is executed instead. Standard local variable declarations are defined as

$$\textbf{let } \underline{x} = \underline{t} \textbf{ in } \alpha \;\equiv\; \textbf{choose } \underline{y} \textbf{ with } \underline{y} = \underline{t} \textbf{ in } \alpha\{\underline{x} \mapsto \underline{y}\} \textbf{ ifnone skip}$$

where $\underline{y}$ are new variables and $\alpha\{\underline{x} \mapsto \underline{y}\}$ denotes the substitution of $\underline{x}$ with $\underline{y}$ in α (the renaming avoids conflicts when $\underline{x}$ is used in $\underline{t}$). Note that **ifnone skip** is never executed here.

3.2. Non-Atomic Semantics of Rules

This section gives a non-atomic semantics to rules: each update and each test of a condition is executed as a separate step. The semantics of rules is based on intervals $I = (I(0), I(1), \ldots)$ of states $I(k) \in S$, which may be finite or infinite. Formally, $I \models \alpha$ expresses that the interval I is a possible execution of α. A finite interval indicates termination. The length of an interval $\#I$ (number of transitions) is in $\mathbb{N} \cup \{\infty\}$. If I is finite it consists of $\#I + 1$ states. In particular, the smallest interval with $\#I = 0$ has one state only.

We lift modification of states to intervals: given a tuple of variables $\underline{x}$ and a sequence of value tuples $\underline{\boldsymbol{a}} = (\underline{a}_0, \underline{a}_1, \ldots)$ of the same length as the interval (where each element $\underline{a}_k$ has the same length as $\underline{x}$), then $I\{\underline{x} \mapsto \underline{\boldsymbol{a}}\}$ is the modified interval, such that $I\{\underline{x} \mapsto \underline{\boldsymbol{a}}\}(k) := I(k)\{\underline{x} \mapsto \underline{a}_k\}$. The semantics of sequential composition of rules $\alpha; \beta$ reduces to the sequential composition of intervals I_1 and I_2, written $I_1 \fatsemi I_2$. For finite I_1 the last state $I_1.\text{last}$ of I_1 must agree with the first one of I_2: $I_1.\text{last} = I_2(0)$. In the result, the duplicate state is removed: $I_1 \fatsemi I_2 := (I_1(0), \ldots, I_1.\text{last}, I_2(1), I_2(2), \ldots)$. If I_1 is infinite the second interval is discarded: $I_1 \fatsemi I_2 := I_1$.

Finite intervals are sometimes written as $(s, \ldots, s')$ where s is the first state and s' is the last state. The notation $(s, \ldots)$ refers to infinite intervals.

Definition 1.

$I \models \underline{loc} := \underline{t}$ iff $I = (s, s')$ and $s' = s\{\underline{loc} \mapsto \underline{t}\}$

$I \models \alpha; \beta$ iff there are I_1, I_2 such that $I_1 \models \alpha$, $I_2 \models \beta$ and $I = I_1 \fatsemi I_2$

$I \models \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta$
iff either $I(0) \models \varepsilon$ and $I \models \textbf{skip}; \alpha$ or $I(0) \not\models \varepsilon$ and $I \models \textbf{skip}; \beta$

$I \models \textbf{choose } \underline{x} \textbf{ with } \varphi \textbf{ in } \alpha \textbf{ ifnone } \beta$
iff either $I(0)\{\underline{x} \mapsto \underline{a}_0\} \models \varphi$ and $I\{\underline{x} \mapsto \underline{\boldsymbol{a}}\} \models \textbf{skip}; \alpha$
for some $\underline{\boldsymbol{a}} = (\underline{a}_0, \underline{a}_1, \ldots)$
or $I \models \textbf{skip}; \beta$ and there are no values $\underline{a}$ with $I(0)\{\underline{x} \mapsto \underline{a}\} \models \varphi$

$I \models \textbf{while } \varepsilon \textbf{ do } \alpha$
iff $I \in \nu(\lambda\, \mathcal{I}.\ \{I_0 \mid$
either $I_0(0) \not\models \varepsilon$ and $I_0 \models \textbf{skip}$
or $\#I_0 = \infty$ and $I_0(0) \models \varepsilon, I_0 \models \textbf{skip}; \alpha$
or $I_0 = I_1 \fatsemi I_2$ with $\#I_1 < \infty, I_1(0) \models \varepsilon, I_1 \models \textbf{skip}; \alpha, I_2 \in \mathcal{I}\})$

Most of the clauses should be intuitive. The **skip**s in the clauses for **if**, **while** and **choose** indicate that evaluating the test is done in a separate step. In the first disjunct of the semantics of choose, the sequence of states $\underline{\boldsymbol{a}}$ captures the values of local variables $\underline{x}$ in the entire interval of α, not just in the first state. The set of runs of a while loop is defined as the greatest fixpoint ν[1] of interval sets $\mathcal{I}$ whose elements I_0 denote different possibilities to execute the loop. Informally, an interval I is a run of the while loop, if it can be split into a (finite or infinite) sequence of adjacent pieces. Each piece I_1 must be finite and execute the loop body (last line $I_1(0) \models \varepsilon, I_1 \models \mathbf{skip}; \alpha$), the only exception being the last interval, when the sequence is finite. This interval may either be a nonterminating (infinite) execution of the loop body (second line of the definition), or it may be one **skip** step, where the loop test evaluates to false (first line of the definition).

3.3. Atomic Semantics of Rules

We define the atomic semantics of rules $[\![\alpha]\!]$ as a relation over the set $S_\perp := S \uplus \{\perp\}$, which augments the set of states with a $\perp$ element to indicate nontermination. From this we derive the semantics of operations in Sec. 4 that additionally check the respective precondition.

Definition 2. *The atomic semantics $[\![\alpha]\!] \subseteq S_\perp \times S_\perp$ of a data type ASM rule α is defined as follows:*

$$
\begin{array}{lll}
(s, s') \in [\![\alpha]\!] & \text{iff either} & (s, \dots, s') \models \alpha \text{ for some finite interval } (s, \dots, s') \\
& \text{or} & (s, \dots) \models \alpha \text{ for some infinite interval } (s, \dots) \text{ and } s' = \perp \\
& \text{or} & s = s' = \perp
\end{array}
$$

The first clause collapses finite runs $(s, \dots, s')$ of α to their first and last state. Infinite runs yield $\perp$ by the second clause. The last line allows to define the semantics of calling two operations sequentially as relational composition: If the first operation does not terminate (gives $\perp$), then attempting to call another operation is not possible and will also give $\perp$.

3.4. Calculus

To formally verify properties of data type ASM rules in KIV we use a weakest-precondition calculus. The calculus defines two program formulas $\langle\!|\alpha|\!\rangle\, \varphi$ and $\langle \alpha \rangle\, \varphi$ as follows, where $s \neq \perp$:

$$
\begin{array}{ll}
s \models \langle\!|\alpha|\!\rangle\, \varphi & \text{iff all } s' \text{ with } (s, s') \in [\![\alpha]\!] \text{ satisfy } s' \neq \perp \text{ and } s' \models \varphi \\
s \models \langle \alpha \rangle\, \varphi & \text{iff there is } s' \neq \perp \text{ with } (s, s') \in [\![\alpha]\!] \text{ and } s' \models \varphi
\end{array}
$$

[1]The greatest fixpoint $\nu(\lambda \mathcal{I}. \{I \mid \varphi(I, \mathcal{I})\})$ is the *union* of all sets $\mathcal{I}$ whose elements satisfy the recursive property φ (which must be monotonic in $\mathcal{I}$). The more commonly used least fixpoint is inadequate here, since it gives the finite executions only.

Formula $\langle\!|\alpha|\!\rangle\,\varphi$ expresses the weakest precondition for rule α to be guaranteed to terminate and to establish postcondition φ (which is often written $\mathbf{wp}(\alpha, \varphi)$ in the literature). Formula $\langle\alpha\rangle\,\varphi$ is from Dynamic Logic [13] and expresses that α has a terminating run after which φ holds. Dynamic Logic writes the weakest liberal precondition $\mathbf{wlp}(\alpha, \varphi)$ as $[\alpha]\,\varphi$, which is equivalent to $\neg\,\langle\alpha\rangle\,\neg\,\varphi$.

Note that in contrast to standard wp-calculus formula φ is not restricted to predicate logic, but may be another program formula. This will be exploited in the proof obligation for simulations (see Theorem 1). The wp-calculus has simple symbolic execution rules for reasoning about rules α (some of these rules can e.g. be found in [14]).

4. Contract Refinement for Data Type ASMs

Our formal definition of a data type ASM $\mathcal{M} = (\mathit{SIG}, \mathit{Ax}, \mathit{Init}, \{\mathtt{Op}_j\}_{j\in J})$ consists of a signature SIG, a set Ax of predicate logic axioms for the static part of the signature, a predicate Init to characterize initial states, and a set of operations for indices $j \in J$. Each operation $\mathtt{Op}_j = (\mathit{pre}_j, \underline{\mathit{in}}_j, \alpha_j, \underline{\mathit{out}}_j)$ consists of a data type ASM rule α_j that describes possible state transitions, provided precondition pre_j holds. It reads input from a tuple $\underline{\mathit{in}}_j$ of input locations, and writes output to a tuple $\underline{\mathit{out}}_j$ of output locations. It may modify local variables, controlled locations and the locations of $\underline{\mathit{out}}_j$. Rules may not have global variables (thus, states of $\mathcal{M}$ are just SIG-Algebras and the values of variables are irrelevant). In concrete code like the ones given in Fig. 3 and Fig. 4 each operation $\mathtt{Op}_j$ has a *name* (instead of using an index j), and is given in the form of $\mathit{name}(\underline{\mathit{in}}_j; \underline{\mathit{out}}_j)$ **pre** pre_j $\{\ \alpha_j\ \}$.

Definition 3. *The atomic semantics* $[\![\mathtt{Op}]\!] \subseteq S_\perp \times S_\perp$ *of a data type ASM operation* $\mathtt{Op} = (\mathit{pre}, \underline{\mathit{in}}, \alpha, \underline{\mathit{out}})$ *extends* $[\![\alpha]\!]$ *in two ways: the input locations are erased (i.e. arbitrary) in the post-state, and any result, including nontermination, is allowed if the precondition does not hold:*

$$\begin{array}{lll} (s, s') \in [\![\mathtt{Op}]\!] & \text{iff either } s \neq \perp, s' \neq \perp \text{ and } (s, s'\{\underline{\mathit{in}} \mapsto \underline{a}\}) \in [\![\alpha]\!] \text{ for some } \underline{a} \\ & \text{or } (s, \perp) \in [\![\alpha]\!] \\ & \text{or } s \not\models \mathit{pre} \text{ and } s' \text{ is arbitrary in } S_\perp \end{array}$$

The intuition for changing the input locations arbitrarily is to allow the environment (e.g., the user of the file system) to set the input locations $\underline{\mathit{in}}$ to new values before the next operation is executed. This coincides with the definition of "firing of updates" of standard ASMs (see [1]), which similarly assigns new values to monitored functions. Non-terminating runs of α are preserved unchanged (in particular $(\perp, \perp) \in [\![\mathtt{Op}]\!]$). The third line of the definition is the standard semantics of a precondition.

Based on the semantics of a single operation we can define runs of a machine $\mathcal{M} = (\mathit{SIG}, \mathit{Ax}, \mathit{Init}, \{\mathtt{Op}_j\}_{j\in J})$ for a finite or infinite sequence $\underline{j} = (j_0, j_1, \ldots)$ of (indices or names of) operation calls. In order to distinguish between an infinite sequence of calls and the divergence of one operation, we allow the states of intervals to end with a sequence of $\perp$.

Definition 4 (Executions & Runs of data type ASMs)**.** *An execution of the call sequence $\underline{j}$ for a data type ASM $\mathcal{M}$ is an interval I, written $I \in exec^{\mathcal{M}}(\underline{j})$, iff $\#I = \#\underline{j}$, and*

$$(I(k), I(k+1)) \in [\![\mathtt{Op}_{j_k}]\!] \quad \text{for all } 0 \leq k < \#j$$

A run, written $I \in runs^{\mathcal{M}}(\underline{j})$, is an execution that starts with an initial state $I(0) \models Init$.

The definition of runs mimics the definition of runs of data types, although we consider both finite and infinite runs. Note that runs may well call an operation with the precondition being false. According to the semantics of one operation (Def. 3) the rest of the run is unpredictable then: either the operation diverges, and the interval ends with a sequence of $\perp$ states, or execution may continue with an arbitrary state.

Refinement between an abstract machine $\mathcal{A} = (SIG^{\mathcal{A}}, Init^{\mathcal{A}}, Ax^{\mathcal{A}}, \{\mathtt{Op}_j^{\mathcal{A}}\}_{j \in J})$ and a concrete machine $\mathcal{C} = (SIG^{\mathcal{C}}, Init^{\mathcal{C}}, Ax^{\mathcal{C}}, \{\mathtt{Op}_j^{\mathcal{C}}\}_{j \in J})$ with the same operation set J is defined relative to a relation $IO \subseteq AS \times CS$ ("input/output correspondence") over the state sets AS and CS of $\mathcal{A}$ and $\mathcal{C}$. Often IO requires identity of input and output locations, but more general cases are possible.

Definition 5. *Correspondence of two intervals $I^{\mathcal{C}}$ and $I^{\mathcal{A}}$ of $\mathcal{C}$ resp. $\mathcal{A}$ ("$I^{\mathcal{C}}$ matches $I^{\mathcal{A}}$ via IO") is defined as*

$$\begin{array}{lll} I^{\mathcal{C}} \sqsubseteq_{IO} I^{\mathcal{A}} & \text{iff } \#I^{\mathcal{C}} = \#I^{\mathcal{A}} \text{ and for all } k \leq \#I^{\mathcal{C}}: & \\ & \text{either } I^{\mathcal{A}}(k) = \perp \ \text{ (and } I^{\mathcal{C}}(k) \text{ is arbitrary)} & \\ & \text{or} \qquad I^{\mathcal{C}}(k) \neq \perp,\ I^{\mathcal{A}}(k) \neq \perp,\ \text{ and } (I^{\mathcal{A}}(k), I^{\mathcal{C}}(k)) \in IO & \end{array}$$

Fig. 6 visualizes two IO-matching runs. In the example, the concrete run diverges after step k, which is matched by the abstract $\perp$. Refinement relative to IO is then defined as follows.

Definition 6 (Data type ASM refinement)**.** *Machine $\mathcal{C}$ refines machine $\mathcal{A}$ relative to IO, written $\mathcal{C} \sqsubseteq_{IO} \mathcal{A}$, if for every call sequence $\underline{j}$ and every $I^{\mathcal{C}} \in runs^{\mathcal{C}}(\underline{j})$ an abstract run $I^{\mathcal{A}} \in runs^{\mathcal{A}}(\underline{j})$ exists, such that $I^{\mathcal{C}} \sqsubseteq_{IO} I^{\mathcal{A}}$ holds. In the common case where $IO = Id$ we omit IO and just write $\mathcal{C} \sqsubseteq \mathcal{A}$.*

The definition allows to refine an abstract run, which calls a diverging operation (for example, one where the precondition is violated), with a terminating run: the state $I^{\mathcal{A}}(k)$ (compare Fig. 6) after the diverging operation (and all subsequent states) will be $\perp$, and match any concrete state. Thus our definition follows the contract approach to data refinement [8].

Proofs of refinement are typically done with forward simulation, composing runs (as those shown in Fig. 6) from commuting diagrams (as shown in Fig. 7).

$$\begin{tikzcd}
as_1 \arrow[r] \arrow[d, dash, "IO"] & as_2 \arrow[r, dashed] \arrow[d, dash, "IO \ldots"] & \perp \arrow[r] \arrow[d, dash, "IO"] & \perp \arrow[r, dashed] \arrow[d, dash, "IO"] & \cdots \\
cs_1 \arrow[r] & cs_2 \arrow[r, dashed] & cs_k \arrow[r] & \perp \arrow[r, dashed] & \cdots
\end{tikzcd}$$

Figure 6: IO-refinement between abstract run $(as_1, \ldots)$ and corresponding concrete run $(cs_1, \ldots)$.

$$\begin{tikzcd}
as_1 \in dom(\mathtt{Op}_j^{\mathcal{A}}) \arrow[r, dotted] \arrow[d, dash, "R"] & \exists\, as_2 \arrow[d, dotted, dash, "R"] \\
cs_1 \arrow[r] & cs_2
\end{tikzcd}$$

Figure 7: Forward simulation R with commuting 1:1 diagrams

Theorem 1 (Forward Simulation)**.** *$\mathcal{C} \sqsubseteq_{IO} \mathcal{A}$ follows from a forward simulation $R \subseteq AS \times CS$ such that $R \subseteq IO$ and*

$$\begin{array}{lll}
\text{Initialization:} & Init^{\mathcal{C}} \subseteq ran(Init^{\mathcal{A}} \lhd R) & \\
\text{Correctness:} & (dom(\mathtt{Op}_j^{\mathcal{A}}) \lhd R) \mathbin{;} [\![\mathtt{Op}_j^{\mathcal{C}}]\!] \subseteq [\![\mathtt{Op}_j^{\mathcal{A}}]\!] \mathbin{;} R & \text{for all } j \in J \\
 & \text{where } dom(\mathtt{Op}_j^{\mathcal{A}}) := \{as \in S \mid (as, \perp) \notin [\![\mathtt{Op}_j^{\mathcal{A}}]\!]\} &
\end{array}$$

Here, $\_ \mathbin{;} \_$ denotes composition of relations (instead of intervals), $\_ \lhd \_$ denotes domain restriction, and *ran* is the range of a relation. The correctness proof obligation is visualized in Fig. 7. A state as_2 has to be found such that the diagram commutes. The restriction $as_1 \in dom(\mathtt{Op}_j^{\mathcal{A}})$ defines the "interesting" cases, where the state as_2 is not $\perp$. Otherwise, $\mathtt{Op}_j^{\mathcal{A}}$ has a non-terminating run, so choosing $as_2 = \perp$ is sufficient for refinement. Note that the correctness condition also enforces termination of $\mathtt{Op}_j^{\mathcal{C}}$ in the interesting case, i.e., cs_2 cannot be $\perp$ (so in particular $pre_j^{\mathcal{C}}(cs_1)$ must hold), since $[\![\mathtt{Op}_j^{\mathcal{A}}]\!] \mathbin{;} R$ does not contain any pair $(as_2, \perp)$. In (the contract version of) data refinement [8], this "applicability" condition is often stated separately.

That a forward simulation is sufficient for refinement is proved as usual by induction over the number of steps that are simulated. The semantic proof obligations for a forward simulation can also be expressed equivalently in the calculus:

$$\begin{array}{lll}
\text{Initialization:} & Init^{\mathcal{C}}(cs) \rightarrow \exists\, as.\ Init^{\mathcal{A}}(as) \wedge R(as, cs) & \\
\text{Correctness:} & \quad pre_j^{\mathcal{A}}(as) \wedge R(as, cs) \wedge \langle\!|\alpha_j^{\mathcal{A}}|\!\rangle\, \mathtt{true} & \\
 & \rightarrow pre_j^{\mathcal{C}}(cs) \wedge \langle\!|\alpha_j^{\mathcal{C}}|\!\rangle\, \langle \alpha_j^{\mathcal{A}} \rangle\, R(as, cs) & \text{for all } j \in J
\end{array}$$

5. Submachines

In this section we define data type ASMs $\mathcal{M}(\mathcal{L})$ that use a submachine $\mathcal{L}$, by extending the syntax and semantics of rules with calls to the operations of $\mathcal{L}$. Proper use of a submachine is subject to certain restrictions that ensure that $\mathcal{M}(\mathcal{L})$ is *independent* of the inner workings of $\mathcal{L}$, i.e., only the inputs and outputs of $\mathcal{L}$ are relevant. As a consequence, the submachine $\mathcal{L}$ can be replaced by a refined machine $\mathcal{K}$ in a compositional way, i.e., $\mathcal{K} \sqsubseteq \mathcal{L}$ implies $\mathcal{M}(\mathcal{K}) \sqsubseteq \mathcal{M}(\mathcal{L})$. Theorem 2 will make the correspondence precise.

For a data type ASM $\mathcal{M} = (SIG, Ax, Init, \{\mathtt{Op}_i\}_{i \in I})$ that calls operations of a submachine $\mathcal{L} = (SIG^{\mathcal{L}}, Ax^{\mathcal{L}}, Init^{\mathcal{L}}, \{\mathtt{Op}^{\mathcal{L}}_j\}_{j \in J})$ the following restrictions apply:

- $\mathcal{M}$ contains $\mathcal{L}$'s signature and axioms: $SIG^{\mathcal{L}} \subseteq SIG$ and $Ax^{\mathcal{L}} \subseteq Ax$. Thereby, a state s of $\mathcal{M}(\mathcal{L})$ can be split into the "local" part ls of $\mathcal{L}$ and the remaining "global" part gs of $\mathcal{M}$, which we write as $s = ls \oplus gs$. Similarly, we write $I^{\mathcal{L}} \oplus I^{\mathcal{M}}$ for the point-wise split of intervals over disjoint signatures.
- Initialization of $\mathcal{M}$ respects initialization of $\mathcal{L}$: $\{ls \mid ls \oplus gs \in Init\} \subseteq Init^{\mathcal{L}}$.
- $\mathcal{M}$ respects information hiding: The signature of $\mathcal{L}$ is never accessed directly by operations of $\mathcal{M}$, i.e., $\mathcal{M}$ can only read and update the signature of $\mathcal{L}$ indirectly via calls to operations of $\mathcal{L}$. The latter means that the local state of $\mathcal{L}$, consisting of the locations in the input, output and controlled signature of $\mathcal{L}$, may not be used in updates, actual call parameters or tests of $\mathcal{M}$ operations.

5.1. Syntax & Semantics of Submachine Calls

Rules α of the machine $\mathcal{M}$ may now contain calls to operations of $\mathcal{L}$. We extend the syntax of rules of Sec. 3.1 with

$$\alpha ::= \ \dots \mid \mathtt{Op}^{\mathcal{L}}_j(\underline{t}; \underline{loc})$$

The call expects the actual inputs $\underline{t}$ in the input locations $\underline{in}^{\mathcal{L}}_j$ of $\mathtt{Op}^{\mathcal{L}}_j$, executes the rule $\mathtt{Op}^{\mathcal{L}}_j$, and finally copies $\underline{out}^{\mathcal{L}}_j$ back to actual outputs $\underline{loc}$, which must be writable locations of $\mathcal{M}$. Within the run of α_j the call to $\mathtt{Op}^{\mathcal{L}}_j$ is considered as one atomic step, except when the submachine call does not terminate ($ls' = \bot$) then I continues with an arbitrary and possibly infinite number of $\bot$ states, as indicated by $\bot^\omega$ below.

In order to record (and later extract) the sequence of submachine calls, we extend the semantics $I \models \alpha$ to $I, \underline{j} \models \alpha$ with an explicit sequence of submachine calls $\underline{j} \in (J \uplus \{\tau\})^\omega$, where J is the index set of $\mathcal{L}$.

Definition 7 (Semantics of submachine calls)**.**

$$\begin{aligned}
&I, \underline{j} \models \mathtt{Op}^{\mathcal{L}}_j(\underline{t}; \underline{loc}) \\
&\quad \textit{iff} \quad ls(\underline{in}^{\mathcal{L}}_j) = [\![\underline{t}]\!](gs) \quad \textit{and} \quad (ls, ls') \in [\![\mathtt{Op}^{\mathcal{L}}_j]\!] \\
&\quad \textit{and } I = \begin{cases} (ls \oplus gs, \quad ls' \oplus gs\{\underline{loc} \mapsto ls'(\underline{out}^{\mathcal{L}}_j)\}) & \textit{if } ls' \neq \bot \\ (ls \oplus gs, \bot^\omega) & \textit{otherwise} \end{cases} \\
&\quad \textit{and } \underline{j} = \begin{cases} (j) & \textit{if } ls' \neq \bot \\ (j, \dots) \textit{ such that } \#\underline{j} = \#I & \textit{otherwise} \end{cases}
\end{aligned}$$

Ordinary assignments in $\mathcal{M}$ appear as stutter steps τ in this call sequence of the submachine, and sequential composition simply concatenates the call

$$\begin{tikzcd}
 & & \overbrace{\hspace{20em}}^{I \models \alpha} & & \\
\mathcal{M}(\mathcal{L})\text{:} & ls\oplus gs \arrow[r, "x_1 \coloneqq t_1"] \arrow[d] & ls\oplus gs' \arrow[r, "\mathtt{Op}_j^{\mathcal{L}}(\underline{t};\underline{loc})"] \arrow[d] & ls'\oplus gs'' \arrow[r, "x_2 \coloneqq t_2"] \arrow[d] & ls'\oplus gs''' \arrow[d] \\
\mathcal{L}\text{:} & ls \arrow[r, dotted, "\tau"] & ls \arrow[r, "[\![\mathtt{Op}_j^{\mathcal{L}}]\!]"] & ls' \arrow[r, dotted, "\tau"] & ls' \\
 & & \underbrace{\hspace{20em}}_{I^{\mathcal{L}} \in exec_{\tau}^{\mathcal{L}}(\tau, j, \tau)} & &
\end{tikzcd}$$

Figure 8: Example execution of a rule α with calls to submachine $\mathcal{L}$ and corresponding execution of $\mathcal{L}$ with stutter steps.

sequences. Thus, I and $\underline{j}$ have the same length. We omit the sequence of submachine calls $\underline{j}$ if it is not needed explicitly.

The above definition[2] ensures that given a run of $\mathcal{M}$ it is always possible to look at the non-atomic semantics of each rule applied and extract a (stuttering) run of $\mathcal{L}$. Fig. 8 exemplifies the extraction by looking at one atomic step from $(ls\oplus gs)$ to $(ls'\oplus gs''')$ in a run of $\mathcal{M}$. Assuming that the rule executed is α, the fine-grained semantics of α gives an interval $I \models \alpha$ that starts with $(ls\oplus gs)$ and ends with $(ls'\oplus gs''')$. In the example the interval consists of three steps. The first and last step are simply assignments to the global $\mathcal{M}$-state gs. The second step is a submachine call that induces a transition from local state ls to ls'. Projecting all states to local states (lower line), we get an execution $I^{\mathcal{L}}$ of $\mathcal{L}$ with explicit stutter steps (indicated by τ) that executes the call sequence (τ, j, τ). Formally, stuttering executions $exec_{\tau}^{\mathcal{L}}(\underline{j})$ (and runs) of $\mathcal{L}$ are defined like ordinary ones, but elements of the call sequence are in $J_{\tau} := J \uplus \{\tau\}$, where the semantics of τ-transitions is just identity.

Proposition 1. *Given an execution $I, \underline{j} \models \alpha$ of a rule α over the signature of $\mathcal{M}$, $I = I^{\mathcal{L}}\oplus I^{\mathcal{M}}$ can be split into the global interval $I^{\mathcal{M}}$ and a stuttering $\mathcal{L}$-execution $I^{\mathcal{L}} \in exec_{\tau}^{\mathcal{L}}(\underline{j})$.*

5.2. Submachine Refinement

In this section we show that refinement is modular in the following sense: Given a machine $\mathcal{M}(\mathcal{L})$ and a refinement $\mathcal{K} \sqsubseteq_{LIO} \mathcal{L}$, then replacing calls to $\mathcal{L}$ in operations of $\mathcal{M}$ with calls to $\mathcal{K}$ gives a machine $\mathcal{M}(\mathcal{K})$ that refines $\mathcal{M}(\mathcal{L})$.

[2] The semantics of submachine calls differs slightly from the one we gave in [9]. In particular, copying input is no longer necessary as it is now assumed to be already there (compare Def. 3), and $\bot$-states now follow a diverging call instead of an infinite sequence of arbitrary states (which likewise implied, that the calling $\mathcal{M}$-operation diverges). Furthermore, we added the sequence of submachine calls to the semantics. These changes simplify the extraction of a stuttering $\mathcal{L}$-run.

$\mathcal{M}(\mathcal{K})$ is defined over the signature $(SIG \setminus SIG^{\mathcal{L}}) \uplus SIG^{\mathcal{K}}$, its initialization is:

$$Init^{\mathcal{M}(\mathcal{K})} := \{ks \oplus gs \mid ls \oplus gs \in Init^{\mathcal{M}(\mathcal{L})} \text{ and } ks \in Init^{\mathcal{K}}\}.$$

The result needs one additional restriction compared to general refinement: LIO is the identity relation over the input and output parameters of $\mathcal{L}$ and $\mathcal{K}$:

$$LIO := \{(ls, ks) \ \mid \ ls(\underline{in}^{\mathcal{L}}) = ks(\underline{in}^{\mathcal{K}}) \ \text{ and } \ ls(\underline{out}^{\mathcal{L}}) = ks(\underline{out}^{\mathcal{K}})\} \tag{1}$$

where $\underline{in}^{\mathcal{L}}$, $\underline{out}^{\mathcal{L}}$ are all input/output locations of $\mathcal{L}$, likewise for $\mathcal{K}$.

For the refinement $\mathcal{M}(\mathcal{K}) \sqsubseteq_{IO} \mathcal{M}(\mathcal{L})$ we can establish identity as the strongest possible relation IO on the two global states:

$$IO := \{(ls \oplus gs, ks \oplus gs') \ \mid \ gs = gs'\}$$

Given these prerequisites we have

Theorem 2 (Compositionality). *$\mathcal{K} \sqsubseteq_{LIO} \mathcal{L}$ implies $\mathcal{M}(\mathcal{K}) \sqsubseteq_{IO} \mathcal{M}(\mathcal{L})$.*

To prove the theorem, one has to note that it is not possible to incrementally construct a run of $\mathcal{M}(\mathcal{L})$ from one of $\mathcal{M}(\mathcal{K})$ by induction over the number of steps done by $\mathcal{M}(\mathcal{K})$. Given $I^{\mathcal{M}(\mathcal{K})} \sqsubseteq_{IO} I^{\mathcal{M}(\mathcal{L})}$ and assuming $I^{\mathcal{M}(\mathcal{K})}$ is a prefix of $I'^{\mathcal{M}(\mathcal{K})}$, then refinement guarantees the existence of $I'^{\mathcal{M}(\mathcal{L})}$ with $I'^{\mathcal{M}(\mathcal{K})} \sqsubseteq_{IO} I'^{\mathcal{M}(\mathcal{L})}$, but not that $I^{\mathcal{M}(\mathcal{L})}$ is a prefix of $I'^{\mathcal{M}(\mathcal{L})}$, so the induction hypothesis is not applicable. Our proof therefore has to consider a full run of $\mathcal{M}(\mathcal{K})$ *as a whole*.

Proof. We have to prove that every run $(s_0, s_1, \ldots) \in runs^{\mathcal{M}(\mathcal{K})}(\underline{i})$ of $\mathcal{M}(\mathcal{K})$ refines some run $(s'_0, s'_1, \ldots)$ of $\mathcal{M}(\mathcal{L})$. Assuming s_k is $ks_k \oplus gs_k$ whenever $s_k \neq \bot$, we construct a run of the form $(ls_0 \oplus gs_0, ls_1 \oplus gs_1, \ldots) \in runs^{\mathcal{M}(\mathcal{L})}(\underline{i})$ (where each state is $\bot$ if $s_k = \bot$) in several steps. The bottom of Fig. 9 depicts this situation.

First, for each step from s_k to s_{k+1} an interval $I_k, \underline{j}_k \models \alpha_{i_k}$ exists, which consists of the steps executed by rule $\mathtt{Op}_{i_k}^{\mathcal{M}(\mathcal{K})}$ and produces submachine calls $\underline{j}_k$ to $\mathcal{K}$. The first state $I_k(0)$ of every interval is s_k (if $s_k = \bot$ then we set $I_k = (\bot)$). The interval is either infinite or ends with $\bot$ whenever $s_{k+1} = \bot$, otherwise it is finite with $I_k.\text{last} = s_{k+1}$.

Second, all I_k can be concatenated to one interval I (depicted at the bottom), similarly all $\underline{j}_k$ yield the sequence of submachine calls $\underline{j}$, and the local states of $\mathcal{K}$ can be projected out to give a stuttering run $I^{\mathcal{K}} \in runs_{\tau}^{\mathcal{K}}(\underline{j})$ of machine $\mathcal{K}$. Since $\mathcal{K} \sqsubseteq_{LIO} \mathcal{L}$, a corresponding run $I^{\mathcal{L}} \in runs_{\tau}^{\mathcal{L}}(\underline{j})$ with $I^{\mathcal{K}} \sqsubseteq_{LIO} I^{\mathcal{L}}$ exists (the two runs in the middle).

Next, the global states of I can be combined with the ones of $I^{\mathcal{L}}$ to give $I' := I^{\mathcal{L}} \oplus I^{\mathcal{M}}$ (shown at the top). Since I and I' have the same length, I' can be split into subintervals $I'_k = I_k^{\mathcal{L}} \oplus I_k^{\mathcal{M}}$ with the same lengths as I_k. Finally, the initial states $I'_k(0)$ for all k are the states s'_k of the run of $\mathcal{M}(\mathcal{L})$ we wanted to construct (their global state is gs_k, whenever $I'_k(0) \neq \bot$).

We show $I'_k, \underline{j}_k \models \alpha_{i_k}\{\mathcal{K} \mapsto \mathcal{L}\}$ for each k by Lemma 1, which proves that the resulting interval is indeed a run of $\mathcal{M}(\mathcal{L})$. □

$$\begin{tikzcd}
I' & ls_0 \oplus gs_0 \arrow[r, dashed] & ls_k \oplus gs_k \arrow[r, thick, "I'_k = I^{\mathcal{L}}_k \oplus I^{\mathcal{M}}_k, \underline{j}_k \models \alpha_{i_k}\{\mathcal{K} \mapsto \mathcal{L}\}"] & ls_{k+1} \oplus gs_{k+1} & \cdots \\
I^{\mathcal{L}} & ls_0 \in \mathit{Init}^{\mathcal{L}} \arrow[r, dashed] & ls_k \arrow[r, "I^{\mathcal{L}}_k"] & ls_{k+1} & \cdots \\
I^{\mathcal{K}} & ks_0 \in \mathit{Init}^{\mathcal{K}} \arrow[r, dashed] & ks_k \arrow[r, "I^{\mathcal{K}}_k"] & ks_{k+1} & \cdots \\
I & ks_0 \oplus gs_0 \arrow[r, dashed] & ks_k \oplus gs_k \arrow[r, "I_k = I^{\mathcal{K}}_k \oplus I^{\mathcal{M}}_k, \underline{j}_k \models \alpha_{i_k}"'] & ks_{k+1} \oplus gs_{k+1} & \cdots
\end{tikzcd}$$

Figure 9: Substitution Lemma

Lemma 1 (Substitution of Submachine Calls)**.** *For all* $I^{\mathcal{K}} \oplus I^{\mathcal{M}}, \underline{j} \models \alpha$ *and* $I^{\mathcal{L}} \in \mathit{exec}^{\mathcal{L}}_{\tau}(\underline{j})$ *with* $I^{\mathcal{K}} \sqsubseteq_{LIO} I^{\mathcal{L}}$*, then* $I^{\mathcal{L}} \oplus I^{\mathcal{M}}, \underline{j} \models \alpha\{\mathcal{K} \mapsto \mathcal{L}\}$ *holds.*

Proof. By structural induction on α. □

Remark: Compositional replacement can be nested. $\mathcal{N}(\mathcal{M}(\mathcal{K})) \sqsubseteq \mathcal{N}(\mathcal{M}(\mathcal{L}))$ for any (proper) outer context $\mathcal{N}$ follows trivially, because IO implies that $\mathcal{M}(\mathcal{L})$ will have the same outputs as $\mathcal{M}(\mathcal{K})$, i.e., satisfies the equivalent of (1) on the outer level.

6. Crashes & Recovery

In this section we consider data type ASMs that exhibit *crashes* during execution. A crash is an event that is triggered asynchronously, aborting the currently executing operation in some intermediate state. In the context of file system verification, this typically means unexpected power loss. Intuitively, a crash erases volatile state (i.e., data in main memory such as caches), but the persistent state remains unchanged; crash safety means that operations have an observably atomic effect with respect to power cuts.

After a crash, a designated *recovery* operation tries to reconstruct the previous situation. In practice, the difficulty is to determine how far the aborted operation has progressed, i.e., whether the operation can be considered to have taken effect or not.

However, the intuitive understanding of crash-safety as observable atomicity is insufficient for realistic systems. In the flash file system, for example, many intermediate layers (i.e., submachines) expose some effect of a crash that cannot be masked by the recovery because of incomplete information (an example is given in the next section, more complex ones are described in [15, 16]). As a consequence, the effect cannot be masked from the specification either, which motivates our approach to integrate crash-safety into refinement: an abstract machine $\mathcal{A}$ has to specify to what extent the corresponding implementation $\mathcal{C}$ must be able to recover from a crash.

6.1. Observable Effects of Crashes in POSIX

On the POSIX level, the effect of a crash and subsequent recovery is to drop all open file handles and to delete all orphaned files. This is formally specified as a binary predicate over pairs of states

$$\begin{aligned}&\mathit{posix\text{-}crash\text{-}recovery}(\mathit{tree}, \mathit{fs}, \mathit{ofh}, \mathit{tree}', \mathit{fs}', \mathit{ofh}')\\ &\quad\leftrightarrow \mathit{tree}' = \mathit{tree} \wedge \mathit{fs}' = \mathit{fs} \setminus \texttt{orphans}(\mathit{tree}, \mathit{fs}) \wedge \mathit{ofh}' = \emptyset\end{aligned}$$

where $\texttt{orphans}(\mathit{tree}, \mathit{fs}) = \{\mathit{ino} \in \mathit{fs} \mid \texttt{links}(\mathit{ino}, \mathit{tree}) = \emptyset\}$, i.e., orphaned files are those that had already been unlinked from the tree at the time of the crash, but were still referenced by some open file handle which now doesn't exist any more. Orphans are now obsolete because no more references to them exist.

Power cuts are treated compositionally in VFS. The crash predicate of VFS is equivalent to the one of AFS, i.e.,

$$\begin{aligned}&\mathit{vfs\text{-}crash}(\mathit{dirs}, \mathit{files}, \mathit{ofh}, \mathit{dirs}', \mathit{files}', \mathit{ofh}')\\ &\quad\leftrightarrow \mathit{afs\text{-}crash\text{-}recovery}(\mathit{dirs}, \mathit{files}, \mathit{dirs}', \mathit{files}')\end{aligned}$$

without specifying a value for ofh' after the crash. This asymmetry is because ofh is now an implementation level variable that is stored in main memory (and is therefore arbitrary after the crash as formalized by Def. 12 below). It is therefore initialized explicitly within the recovery procedure of VFS in order to establish the requirement imposed by *posix-crash-recovery*:

$$\texttt{vfs\_recover}()\ \{\ \mathit{ofh} \coloneqq \emptyset\ \}$$

The effect specified by POSIX is matched on the AFS level, for a suitable definition of `orphans` (that commutes with the simulation relation).

$$\begin{aligned}&\mathit{afs\text{-}crash\text{-}recovery}(\mathit{dirs}, \mathit{files}, \mathit{dirs}', \mathit{files}')\\ &\quad\leftrightarrow \mathit{dirs}' = \mathit{dirs} \wedge \mathit{files}' = \mathit{files} \setminus \texttt{orphans}(\mathit{dirs}, \mathit{files})\end{aligned} \tag{2}$$

6.2. Atomicity of Operations

The basic idea is that the non-atomic semantics of a rule $I \models \alpha$ defines the states $I(k)$ in which crashes need to be considered. An example is shown in Fig. 10 where an operation of a machine $\mathcal{M}(\mathcal{L})$ executes several steps. Disregarding the submachine call for a moment, an example crash occurs in state after two steps, followed by a recovery to another state represented by the circle at the top right. The crash semantics defined in terms of intervals $I \models \alpha$ will allow us to reason about such intermediate states. In order to specify the properties of such states, we will use refinement, just as we do for ordinary, final states.

Extending this notion to submachines with crashes raises the question about atomicity of calls. To what extent can they be treated atomically? In Fig. 10, the submachine operation at the bottom has some intermediate states (gray). Given that $\mathcal{L}$ is an *abstract* machine, i.e., one that is refined further, it makes

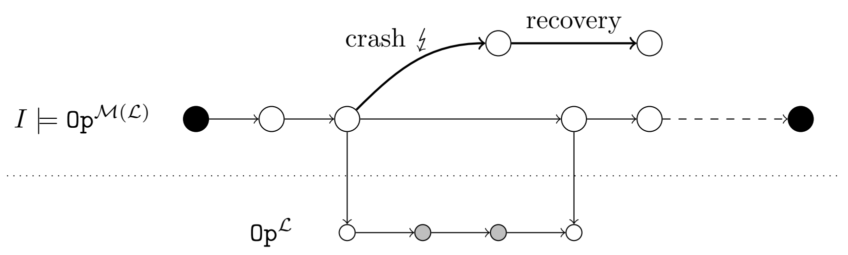
crash ↯
recovery
$I \models \mathtt{Op}^{\mathcal{M}(\mathcal{L})}$
$\mathtt{Op}^{\mathcal{L}}$

Figure 10: Different degrees of atomicity of crash behavior of an operation of machine $\mathcal{M}(\mathcal{L})$. The states denoted by circles are generated by the semantics from Sec. 3.2. $\mathcal{M}^{\blacksquare}(\mathcal{L}^{\blacksquare})$ considers the black states only, $\mathcal{M}^{\square}(\mathcal{L}^{\blacksquare})$ adds the white states to the behavior and $\mathcal{M}^{\square}(\mathcal{L}^{\square})$ additionally considers the gray states. At the top, an example crash in an intermediate state and subsequent recovery is shown. The corresponding states at the top are generated by the crashing semantics as defined in Sec. 6.4.

sense to interpret $\mathcal{L}$-calls as atomic, since proofs for crashed runs of $\mathcal{M}(\mathcal{L})$ will be simpler. Consequently, only crashes within $\mathcal{L}$ in the white states should be considered. In contrast, if the submachine corresponds to an implementation, *all* intermediate states (white and gray) are relevant for the analysis, because that is what will eventually run in reality. We capture these different views by two semantics for crash-behavior:

A machine $\mathcal{M}^{\square}$ with "white-box" crash behavior permits crashes any time during the execution of its operations. Under certain conditions, such a machine can be reduced to a machine $\mathcal{M}^{\blacksquare}$ with "black-box" crash behavior: only states in between operations are relevant in order to study crashes. Black-box crash behavior of submachines $\mathcal{L}^{\blacksquare}$ is what makes it possible to treat calls atomically. In the following, let $\mathcal{M}^{◩}$ denote a machine with either crash behavior $\mathcal{M}^{\square}$ or $\mathcal{M}^{\blacksquare}$.

A key feature is that refinement remains compositional with respect to submachines. The distinction into black-box and white-box semantics gives rise to different composition patterns with increasing degree of atomicity (cf. Fig. 10): Machines $\mathcal{M}^{\square}(\mathcal{L}^{\square})$ consider crashes at any point in time during execution, even in the middle of submachine calls. Such machines correspond to the system that will finally run, i.e., the system whose correctness we are ultimately interested in. Machines $\mathcal{M}^{\square}(\mathcal{L}^{\blacksquare})$ view operations of $\mathcal{L}$ atomically, however, it is still possible to have a crash in the middle of an $\mathcal{M}$-operation. Machines $\mathcal{M}^{\blacksquare}(\mathcal{L}^{\blacksquare})$ consider crashes between $\mathcal{M}$-operations only. From a verification point-of-view such machines are much simpler, since it is possible to reason about their behavior in an entirely atomic setting. Technically, this means that weakest precondition calculus, as presented in Sec. 3.4, is sufficient for proofs of their correctness.

Generally, we wish to switch from a white-box to a black-box view whenever possible, because a formal proof for the latter has to consider *significantly* less states. The two main theorems are:

- A submachine with crashes can be substituted in a context, e.g., $\mathcal{K}^{\blacksquare} \sqsubseteq \mathcal{L}^{\blacksquare}$ implies $\mathcal{M}^{\square}(\mathcal{K}^{\blacksquare}) \sqsubseteq \mathcal{M}^{\square}(\mathcal{L}^{\blacksquare})$ (Theorem 4), and
- $\mathcal{M}^{\square}(\mathcal{L}^{\blacksquare}) \equiv \mathcal{M}^{\blacksquare}(\mathcal{L}^{\blacksquare})$ holds under certain conditions (Theorem 5).

6.3. Application to the File System

In general, we are interested in the correctness of the machine (omitting several intermediate machines for clarity)

$$\mathcal{C} := \mathit{VFS}^{\square}(\mathit{FS}^{\square}(\mathit{HW}^{\blacksquare})),$$

since this is the final code that will run (cf. Fig. 1). We assume that our atomic specification $\mathit{HW}^{\blacksquare}$ of the flash driver already captures the full crash behavior correctly [15]. The correctness of the entire system is expressed as the refinement $\mathcal{C} \sqsubseteq \mathit{POSIX}^{\blacksquare}$. Thus, from the perspective of the user of our POSIX-compliant file system either an operation took effect in its entirety before a crash or the operation had no effect.

We start with the innermost machine $\mathit{FS}^{\square}(\mathit{HW}^{\blacksquare})$ with full crash behavior and reduce it to $\mathit{FS}^{\blacksquare}(\mathit{HW}^{\blacksquare})$. The reason why $\mathit{FS}^{\square}(\mathit{HW}^{\blacksquare}) \equiv \mathit{FS}^{\blacksquare}(\mathit{HW}^{\blacksquare})$ holds is twofold: First, the entire state of $\mathit{FS}^{\square}$ is considered to be in RAM and is therefore arbitrary after a crash. The state of $\mathit{HW}^{\blacksquare}$ on the other hand is unchanged by a crash. Second, all operations of $\mathit{HW}^{\blacksquare}$ can always fail without changing the stored data (but may signal an error code by setting some output parameter). This models that the flash driver can always run out of memory for some memory allocation (for e.g. buffers) or simply that the flash hardware could not handle the request. From an execution I of an operation of $\mathit{FS}^{\square}(\mathit{HW}^{\blacksquare})$ that crashed in the middle, we can construct an execution of $\mathit{FS}^{\blacksquare}(\mathit{HW}^{\blacksquare})$ by extending I just by executing the operation to the end and choosing that every submachine call to $\mathit{HW}^{\blacksquare}$ should fail without modifying the persistent data. The complete execution then crashes to the same state as the incomplete one.

This allows us to refine the easier system $\mathit{FS}^{\blacksquare}(\mathit{HW}^{\blacksquare})$ to $\mathit{AFS}^{\blacksquare}$

$$\mathit{FS}^{\square}(\mathit{HW}^{\blacksquare}) \sqsubseteq \mathit{FS}^{\blacksquare}(\mathit{HW}^{\blacksquare}) \sqsubseteq \mathit{AFS}^{\blacksquare}. \tag{3}$$

Since submachines with crashes are compositional, we can substitute $\mathit{AFS}^{\blacksquare}$ for $\mathit{FS}^{\square}(\mathit{HW}^{\blacksquare})$ in the context of VFS. Thus, we have proven

$$\mathit{VFS}^{\square}(\mathit{FS}^{\square}(\mathit{HW}^{\blacksquare})) \sqsubseteq \mathit{VFS}^{\square}(\mathit{FS}^{\blacksquare}(\mathit{HW}^{\blacksquare})) \sqsubseteq \mathit{VFS}^{\square}(\mathit{AFS}^{\blacksquare}).$$

Now we again have a machine with intermediate crashes with a submachine. However, it now does not necessarily hold that every operation of $\mathit{AFS}^{\blacksquare}$ has a run that leaves the state unchanged, because there is no longer a clear distinction between in-RAM and persistent state in the $\mathit{AFS}^{\blacksquare}$ submachine. Therefore, we generalize the property: An operation is crash-neutral, if it has an execution that can be reverted by a crash. This property is formalized in Sec. 7 and we prove that this is sufficient to construct a complete execution with subsequent crash to a partial, crashing execution (as we did above). The operations of the hardware model are trivially crash-neutral. This also holds for $\mathit{AFS}^{\blacksquare}$.

Therefore, we can complete the refinement tower, by repeating the strategy that was used to establish (3) and get:

$$\mathit{VFS}^{\square}(\mathit{AFS}^{\blacksquare}) \equiv \mathit{VFS}^{\blacksquare}(\mathit{AFS}^{\blacksquare}) \sqsubseteq \mathit{POSIX}^{\blacksquare}$$

Note that we need to reason about machines with crashes at the end of operations only and about the crash-neutrality of each submachine, both of which can be expressed in the calculus.

6.4. Semantics of Crashes

Definition 8 (Data type ASM with crash behavior)**.** *A data type ASM with crash behavior* $\mathcal{M}^{\text{◩}} = (\mathit{SIG}, \mathit{Ax}, \mathit{Init}, \{\mathtt{Op}_j\}_{j \in J}, \mathit{Cr})$ *has an additional static predicate* $\mathit{Cr} \subseteq S \times S$ *describing possible state transitions of the dynamic part of the signature triggered by a power cut. Immediately afterwards (and only then) a designated operation* $\mathtt{Op}_{\mathrm{rec}}$ *with index* $\mathrm{rec} \in J$ *is called implicitly in order to restore a consistent state.*

We define the semantics of machines $\mathcal{M}^{\square}$ and $\mathcal{M}^{\blacksquare}$ in terms of a modified atomic semantics of their rules, $[\![\alpha]\!]^{\square}$ and $[\![\alpha]\!]^{\blacksquare}$ respectively. In analogy to non-termination $\bot$, the state space is extended with states $s_{\lightning} \in S_{\lightning}$ that signal a crashed state where $S_{\lightning} := \{s_{\lightning} \mid s \in S\}$. The $\lightning$ is merely an annotation, that indicates that the state s cannot be used by any operation except recovery.

Crashed states imply that the currently running operation is aborted and subsequent steps are not possible. Sequential composition of intervals is adapted so that $I_0 \fatsemi I_1 := I_0$ whenever $I_0.\mathrm{last} = s_{\lightning}$ is a crashed state, i.e., the remainder I_1 is not executed.

Definition 9 (White-box crash semantics of rules)**.**

$$\begin{aligned}
(s, s') \in [\![\alpha]\!]^{\square} \quad & \text{iff either } (s, s') \in [\![\alpha]\!] \\
& \quad \text{or for some } s_0, s_1, I: \\
& \qquad (s, \ldots, s_0) \fatsemi I \models \alpha \text{ and } (s_0, s_1) \in \mathit{Cr} \text{ and } s' = s_{1\lightning} \\
& \quad \text{or } (s, \ldots, s_{1\lightning}) \models \alpha \text{ and } s' = s_{1\lightning}
\end{aligned}$$

The first clause permits an uncrashed execution. The second clause splits the fine-grained semantics of α into a finite prefix $(s, \ldots, s_0)$ and a remainder I (that is ignored). Assuming that s_0 is a normal state the effect of the crash Cr is applied to get to s_1 and the $\lightning$-marker is set. The third clause propagates crashes that have occurred during the submachine calls (see Def. 11).

Definition 10 (Black-box crash semantics of rules)**.**

$$\begin{aligned}
(s, s') \in [\![\alpha]\!]^{\blacksquare} \quad & \text{iff either } (s, s') \in [\![\alpha]\!] \\
& \quad \text{or for some } s_0, s_1: \\
& \qquad (s, s_0) \in [\![\alpha]\!] \text{ and } (s_0, s_1) \in \mathit{Cr} \text{ and } s' = s_{1\lightning} \\
& \quad \text{or } (s, s_1) \in \mathit{Cr} \text{ and } s' = s_{1\lightning}
\end{aligned}$$

The second clause introduces crashes after *terminating* executions of α. The third clause permits immediate crashes before α has even started. In contrast to Def. 9 it is sufficient to refer to the atomic semantics of α. Note that $s_0 \in S$

is neither $\bot$ nor crashed since $Cr \subseteq S \times S$. The black-box semantics is a subset of the white-box one, i.e., $[\![\alpha]\!]^{\blacksquare} \subseteq [\![\alpha]\!]^{\square}$.

The atomic semantics of operations with crashes $[\![\texttt{Op}_j]\!]^{◩}$ for $j \in J \setminus \{\text{rec}\}$ (except recovery) simply refers to the corresponding semantics of its rule $[\![\alpha]\!]^{◩}$ (analogously to Def. 3). Since $[\![\alpha]\!]^{◩}$ does not contain any transitions starting in a crashed state, the same holds for $[\![\texttt{Op}_j]\!]^{◩}$, i.e., $[\![\texttt{Op}_j]\!]^{◩} \subseteq S_\bot \times (S_\bot \uplus S_{\lightning})$. This captures the notion that normal operations are prohibited in crashed states. In contrast, the recovery operation $\texttt{Op}_{\text{rec}} = (\texttt{true}, \langle\rangle, \alpha_{\text{rec}}, \underline{out})$ is only run in a crashed state. Its atomic semantics $[\![\texttt{Op}_{\text{rec}}]\!]^{◩} \subseteq (S_{\lightning} \uplus \{\bot\}) \times S_\bot$ is defined by

$$\begin{aligned}(s, s') \in [\![\texttt{Op}_{\text{rec}}]\!]^{◩} \quad & \text{iff } s = \bot \\ & \text{or there is a state } \tilde{s} \in S \text{ with } \tilde{s}_{\lightning} = s \text{ and } (\tilde{s}, s') \in [\![\alpha_{\text{rec}}]\!],\end{aligned}$$

i.e., if s is some crashed state $\tilde{s}_{\lightning}$, then the crash marker is removed and the recovery rule is called.

To simplify the discussion, we have excluded crashes during recovery by referring to the normal non-crashing atomic semantics of α here. However, the assumptions of Theorem 5 are sufficient to guarantee crash-safety even then. We also assume that the recovery operation has no input parameters and its precondition is just `true`.

The executions and runs of machines $\mathcal{M}^{◩}$ are defined in terms of $[\![\texttt{Op}_j]\!]^{◩}$ just like in Def. 4. As a consequence crashed states are *exposed* in the runs, reflecting the fact that crashes should be observable. A single transition in a run can be a normal execution of a callable operation, or a crashing execution of such an operation resulting in some state $\tilde{s}_{\lightning}$, in which the subsequent step must be recovery.

6.5. Crash-Safe, Compositional Submachine Refinement

Refinement $\mathcal{C}^{◩} \sqsubseteq_{IO_{\lightning}} \mathcal{A}^{◩}$ between two machines with crash behavior needs to be adapted slightly in comparison to the conditions given in Sec. 4. The relation IO is extended to crashed states as follows: $(as, cs) \in IO_{\lightning}$ iff $(as, cs) \in IO$ are normal states, or both as and cs are crashed (without further constraints).

The proof for forward simulation proceeds by discerning whether the concrete operation $\texttt{Op}^{\mathcal{C}}$ exhibits a crash or not. If there is a crash, we do not construct a 1:1 diagram as shown in Fig. 7, but instead a 2:2 diagram that includes recovery as the next step. A crash is therefore never considered in isolation, and it is not necessary to extend the simulation relation R to crashed states. The extra proof obligations that support this reasoning can be factored out as follows:

Theorem 3 (Forward Simulation with Crashes)**.** *$\mathcal{C}^{◩} \sqsubseteq \mathcal{A}^{◩}$ follows from a forward simulation that satisfies the conditions of Theorem 1 (for the index set $J \setminus \{\text{rec}\}$) and additionally for all $j \in J \setminus \{\text{rec}\}$:*

$$\text{Crash:} \quad (dom(\texttt{Op}^{\mathcal{A}}_j) \lhd R) \fatsemi [\![\texttt{Op}^{\mathcal{C}}_j]\!]^{◩} \fatsemi [\![\texttt{Op}^{\mathcal{C}}_{\text{rec}}]\!]^{◩} \subseteq [\![\texttt{Op}^{\mathcal{A}}_j]\!]^{◩} \fatsemi [\![\texttt{Op}^{\mathcal{A}}_{\text{rec}}]\!]^{◩} \fatsemi R$$

The proof that the extended forward simulation establishes refinement for the modified setting is again by induction over the number of steps executed. Note that we have defined $IO_{\lightning}$ to hold for any pair of crashed states in the middle of the 2:2 diagram, so there is nothing to prove for these.

A direct proof for the new conditions of Theorem 3 for the refinements of our case study is however difficult in the white-box setting, because it is necessary to reason about all intermediate states of the execution of $[\![\mathtt{Op}_j^{\mathcal{C}}]\!]^{\square}$. The abstract operation $[\![\mathtt{Op}_j^{\mathcal{A}}]\!]^{\square}$ is less critical, because we can *choose* to treat it atomically by witnessing whole executions only.

For a black-box refinement $\mathcal{C}^{\blacksquare} \sqsubseteq \mathcal{A}^{\blacksquare}$ the theorem yields the following simple proof obligations, in addition to “Initialization” and “Correctness” from Sec. 4.

$$\begin{array}{ll}\text{Recovery:} & R(as, cs) \wedge (cs, cs') \in Cr^{\mathcal{C}} \\ & \rightarrow \langle\!|\mathtt{Op}_{\mathrm{rec}}^{\mathcal{C}}(; cs')|\!\rangle\, (\, \exists as'.\ (as, as') \in Cr^{\mathcal{A}} \wedge \langle \mathtt{Op}_{\mathrm{rec}}^{\mathcal{A}}(; as')\rangle\, R(as', cs')\,)\end{array}$$

Next, we describe how the desired behavior of the different composition patterns $\mathcal{M}^{\square}(\mathcal{L}^{\square})$, $\mathcal{M}^{\square}(\mathcal{L}^{\blacksquare})$ and $\mathcal{M}^{\blacksquare}(\mathcal{L}^{\blacksquare})$, which we have outlined in the introduction to this section, is established.

Proper use $\mathcal{M}^{◩}(\mathcal{L}^{◩})$ has an extra condition: the crash predicate Cr of $\mathcal{M}$ can be split into the effects $Cr^{\mathcal{L}}$ on the local state of $\mathcal{L}$ and a predicate $Cr^{\mathcal{M}}$ on the global state of $\mathcal{M}$:

$$Cr := \{(ls \oplus gs, ls' \oplus gs') \mid (ls, ls') \in Cr^{\mathcal{L}} \text{ and } (gs, gs') \in Cr^{\mathcal{M}}\}$$

In a white-box context $\mathcal{M}^{\square}$, a crash can occur either in $\mathcal{M}$-steps or in $\mathcal{L}$-calls. For the former we simply apply Cr (compare second case in Def. 9). For the latter, by the semantics of $\mathcal{L}^{◩}$-operations, the resulting state $ls'_{\lightning}$ has already been modified by $Cr^{\mathcal{L}}$ and only $Cr^{\mathcal{M}}$ must be applied (lifting the marker $\lightning$).

The semantics of submachine calls is therefore (extending Def. 7):

Definition 11 (Semantics of submachine calls with crashes ($j \neq \mathrm{rec}$))**.**

$$\begin{array}{l} I \models \mathtt{Op}_j^{\mathcal{L}^{◩}}(\underline{t}; \underline{loc}) \\ \quad \text{iff} \quad ls(\underline{in_j^{\mathcal{L}}}) = [\![\underline{t}]\!](gs) \quad \text{and} \quad (ls, ls') \in [\![\mathtt{Op}_j^{\mathcal{L}}]\!]^{◩} \\ \quad \text{and } I = \begin{cases} (ls \oplus gs, ls' \oplus gs\{\underline{loc} \mapsto ls'(\underline{out_j^{\mathcal{L}}})\}) & \text{if } ls' \neq \bot \text{ and } ls' \notin S_{\lightning} \\ (ls \oplus gs, (\tilde{ls} \oplus gs')_{\lightning}) & \text{if } ls' = \tilde{ls}_{\lightning} \text{ for some } \tilde{ls} \\ & \text{and } (gs, gs') \in Cr^{\mathcal{M}} \\ (ls \oplus gs, \bot^{\omega}) & \text{if } ls' = \bot \end{cases} \end{array}$$

This semantics also works within a black-box context $\mathcal{M}^{\blacksquare}$: Def. 10 filters out crashes in calls to submachines by the condition that only executions of the rule α leading to normal states are accepted. The additional case for submachine calls given above is irrelevant for black-box runs.

Analogously to Theorem 2, the following compositionality result holds:

Theorem 4. *Let $\mathcal{C} \in \{\mathcal{K}^{\blacksquare}, \mathcal{K}^{\square}\}$ and $\mathcal{A} \in \{\mathcal{L}^{\blacksquare}, \mathcal{L}^{\square}\}$ be data type ASMs with crash behavior, then $\mathcal{C} \sqsubseteq \mathcal{A}$ implies $\mathcal{M}^{\square}(\mathcal{C}) \sqsubseteq \mathcal{M}^{\square}(\mathcal{A})$ and $\mathcal{M}^{\blacksquare}(\mathcal{C}) \sqsubseteq \mathcal{M}^{\blacksquare}(\mathcal{A})$.* $\square$

7. Crash Reductions

In this section we formulate two assumptions, which in practice allow us to only consider crashes at the end of operations, i.e., we can prove $\mathcal{M}^{\square}(\mathcal{L}^{\blacksquare}) \equiv \mathcal{M}^{\blacksquare}(\mathcal{L}^{\blacksquare})$ under these assumptions. The basic idea is that for every execution of an operation that crashes in the middle resulting in state s, we can construct a complete execution with a crash at the end, that still yields s. Thus, we can *move* or *postpone* a crash to the end of an operation without any visible effect.

The first assumption restricts the crash predicate of $\mathcal{M}^{◩}$: It must be such that the entire state of $\mathcal{M}$ without the submachine is arbitrary. This allows us to move a crash over one step (other than a submachine call) of an $\mathcal{M}$-rule α while still resulting in the same state after the crash:

Definition 12 (Unrestricted $\mathcal{M}^{◩}$ Crash)**.** *The crash of $\mathcal{M}^{◩}$, which properly uses the submachine $\mathcal{L}$, is unrestricted iff $Cr^{\mathcal{M}}$ satisfies*

$$(gs, gs') \in Cr^{\mathcal{M}} \qquad \text{for all } gs, gs'.$$

This assumption corresponds to the intuition that the state of $\mathcal{M}^{◩}$ is in RAM and therefore lost during a crash. For the VFS model of the case study this is satisfied as shown in Sec. 6.1, only the open file handles *ofh* are part of the state of VFS, the directory structure *dirs* and the file content *files* are part of the submachine AFS.

Complementarily, the second assumption about each operation of the submachine $\mathcal{L}^{◩}$ allows us to move a crash over a submachine call:

Definition 13 (Crash-Neutral)**.** *An operation $\mathtt{Op}_i^{\mathcal{L}}$ is crash-neutral iff*

$$pre_i^{\mathcal{L}} \lhd [\![Cr^{\mathcal{L}}]\!] \subseteq [\![\mathtt{Op}_i^{\mathcal{L}}]\!] \,⨾\, [\![Cr^{\mathcal{L}}]\!].$$

Crash-neutrality is trivially implied by $[\![\mathtt{Op}_i^{\mathcal{L}}]\!] \subseteq Id$. This simpler condition is satisfied is satisfied by all operations of the hardware, which are typically modeled as rules of the form

```
flash_op($\underline{in}$; $err$)
    { $flash := f(\underline{in}, flash)$; $err :=$ ESUCCESS } or { $err :=$ EFAIL }
```

where the failure case witnesses the crash neutral run.

There are exceptions, though, which do require the generality of this definition. On the AFS level, for example, the error handling that is added to the operations shown in Fig. 4 follow the scheme above for `afs_unlink` (which therefore satisfies $[\![\mathtt{afs\_unlink}]\!] \subseteq Id$).

However, `afs_evict` is an exception since never fails (roughly speaking, the corresponding unlink operation dropping the last link has already succeeded previously, and evict operates on RAM data exclusively). To prove that `afs_evict` is crash-neutral for the case where $\mathtt{links}(ino, dirs) = \emptyset$, after substituting Fig. 4 and (2) into Def. 13 it remains to be shown that

$$files \setminus \mathtt{orphans}(dirs, files) \subseteq files \setminus \{ino\} \setminus \mathtt{orphans}(dirs, files)$$

$$\begin{tikzcd}
ls_0 \arrow[rr, "Cr^{\mathcal{L}}"] \arrow[dr, dotted, "\mathtt{Op}_i^{\mathcal{L}}"'] & & ls_1 \\
& \exists ls_2 \arrow[ur, dotted, "Cr^{\mathcal{L}}"'] &
\end{tikzcd}$$

Figure 11: Crash-Neutrality guarantees the existence of ls_2

$$\begin{tikzcd}
& & & s'' \\
s_0 \arrow[r, "\alpha"] & s_1 \arrow[urr, bend left, "\text{completion } I"] \arrow[rr, "Cr"] \arrow[dr, dotted, "\text{next step of } \alpha"'] & & s'_{\lightning} \\
& & \exists s_2 \arrow[ur, dotted, "Cr"] \arrow[r, dotted, "\text{completion } I'"'] & \exists s_n
\end{tikzcd}$$

Figure 12: Reduction to Complete Runs

This holds since $\mathtt{links}(\mathit{ino}, \mathit{dirs}) = \emptyset$ implies $\mathit{ino} \in \mathtt{orphans}(\mathit{dirs}, \mathit{files})$.

Fig. 11 shows an arbitrary transition of a crash from ls_0 to ls_1. If $\mathtt{Op}_i^{\mathcal{L}}$ is crash-neutral then there is a state ls_2 that is reachable through a submachine call to $\mathtt{Op}_i^{\mathcal{L}}$ and also yields ls_1 if we postpone the crash until after the call. This allows us to complete a crashed run of $\mathcal{M}^{\square}(\mathcal{L}^{\blacksquare})$ until the operation terminates, by extending the crashed run with these transitions of crash-neutral operations.

Lemma 2. *If all operations of $\mathcal{L}$ are crash-neutral, the $\mathcal{M}^{◪}(\mathcal{L}^{\blacksquare})$ crash is unrestricted, and all runs of $\mathtt{Op}_j^{\mathcal{M}} = (\mathit{pre}_j, \underline{\mathit{in}}_j, \alpha, \underline{\mathit{out}}_j)$ terminate within pre_j, then $[\![\mathtt{Op}_j^{\mathcal{M}}]\!]^{\square} = [\![\mathtt{Op}_j^{\mathcal{M}}]\!]^{\blacksquare}$ holds.*

Proof. Given $(s_0, s'_{\lightning}) \in [\![\alpha]\!]^{\square}$, we have by definition of $[\![\alpha]\!]^{\square}$ the case of a crashing run $(s_0, \ldots, s_1) \fatsemi I \models \alpha$ with $(s_1, s') \in \mathit{Cr}$ and I finite. The case of a crashing submachine call is not possible here, and the case without crash is trivial. Fig. 12 depicts this situation. We incrementally extend $(s_0, \ldots, s_1)$ to a complete run of α ending in a state s_n with $(s_n, s') \in \mathit{Cr}$, in which case we are done.

If the first transition in I is a submachine call to $\mathtt{Op}_i^{\mathcal{L}}$, we know that the precondition holds, since otherwise $\mathtt{Op}_j^{\mathcal{M}}$ has an infinite run. Since $\mathtt{Op}_i^{\mathcal{L}}$ is crash-neutral, there is a state ls_2 with $(ls_1, ls_2) \in [\![\mathtt{Op}_i^{\mathcal{L}}]\!]$ and $(ls_2, ls') \in Cr^{\mathcal{L}}$ (cf. Fig. 11). We extend the partial run with the state $s_2 := ls_2 \oplus gs_1$. Note that $(s_2, s') \in \mathit{Cr}$ holds. Furthermore, we know that a finite completion I' of the new run exists, because rules can not get stuck and all runs of $\mathtt{Op}_j^{\mathcal{M}}$ terminate.

Otherwise, the first transition of I leaves the state of $\mathcal{L}$ unchanged. We extend the partial run with $s_2 := I(1)$ and the completion I' is $(I(1), \ldots, I.\text{last})$.

In both cases we have extended the partial run of α one step while still being able to crash to the same state s'. We repeat this process until we have a complete run of α that crashes to s'. If this process does not terminate, we have constructed an infinite run of α, contradicting the assumptions. $\square$

We lift this property to the entire machine (via Theorem 3):

Theorem 5. *If all operations of $\mathcal{L}$ are crash-neutral, the $\mathcal{M}^{◪}(\mathcal{L}^{\blacksquare})$ crash is unrestricted, and all runs of all operations of $\mathcal{M}$ terminate within their precondition, then $\mathcal{M}^{\square}(\mathcal{L}^{\blacksquare}) \equiv \mathcal{M}^{\blacksquare}(\mathcal{L}^{\blacksquare})$ holds.* $\square$

The conditions of this theorem yield the following additional proof obligations in the calculus (where J is the index set of $\mathcal{M}$ and K the one of $\mathcal{L}$):

$$\begin{array}{lll}
\text{Termination:} & pre_j^{\mathcal{M}} \rightarrow \langle\!|\alpha_j^{\mathcal{M}}|\!\rangle\, \texttt{true} & \text{for all } j \in J \\
\mathcal{M}^{\text{◩}}(\mathcal{L}^{\blacksquare})\text{-Crash:} & Cr(ls{\oplus}gs, ls'{\oplus}gs') \leftrightarrow Cr^{\mathcal{L}}(ls, ls') & \\
\text{Crash-Neutral:} & pre_k^{\mathcal{L}}(ls) \wedge Cr^{\mathcal{L}}(ls, ls') & \\
 & \rightarrow \langle\alpha_k^{\mathcal{L}}\rangle\,(\; Cr^{\mathcal{L}}(ls, ls')\;) & \text{for all } k \in K
\end{array}$$

Note that ls in the precondition of "Crash-Neutral" refers to the state before executing $\alpha_k^{\mathcal{L}}$, whereas ls after $\langle\alpha_k^{\mathcal{L}}\rangle$ refers to the post-state.

8. Related Work

The first part of this section summarizes related work to refinement and submachines. A more detailed description has been given in [9]. The second part discusses related work with regard to crash-safety and file systems.

8.1. ASMs & Submachines

In general our approach is based on iterated refinement, following the idea of ASM refinement [2]. The specific instance of refinement defined here is based on data refinement [17], in particular the contract-based approach of Z [8]. It can be viewed as an adaption of this approach to the setting of ASMs. We prefer the operational style of ASM rules over the relational style of Z operations, since ASMs can be executed and directly translated to code. Nevertheless, our atomic semantics (Def. 3) of ASM operations parallels the contract embedding of Z relations into states with bottom, except that we do not add $\{\perp\} \times S_\perp$, but just $\{\perp\} \times \{\perp\}$ to preserve the meaning of $\perp$ as "nontermination" (not "unspecified"). [18] argues that for both embeddings the same refinements are correct, in particular our simulation proof obligations are those of Z refinement.

It is a folklore theorem of data refinement that proof obligations for individual operations are sufficient to allow substitution of abstract with concrete operations in any reasonable context, i.e., one that does not access the local state of operations. Our formal proof of Theorem 2 shows that ASM rules are one suitable context. We are not aware of formal proofs that propagate refinement expressed in terms of submachine runs to the context. In [19] analogous results are shown that simulations propagate (Theorems 4.10 and 9.5). This is simpler since an incremental construction over the number of steps is possible while we need to consider the run as a whole (cf. Theorem 2).

The refinement concept discussed here differs from our earlier formalisation [20, 21], and from Event-B [22] in that it uses *preconditions*, not *guards* (the earlier B formalism [23] had both preconditions and guards). Whether one needs one or the other concept is application dependent: when rules are called by the environment, the precondition approach is appropriate; if the machine itself chooses a rule, then the guard interpretation is appropriate.

The definition given here is on the one hand more liberal than the one in [21], as it allows one to implement a diverging operation on the abstract level with any run on the concrete level. On the other hand it is more strict, as it forbids general $m\!:\!n$ diagrams where $m > 1$ abstract operations are implemented with n concrete ones, since the environment cannot be forced to call a specific sequence of m operations.

With respect to ASMs, our syntax only uses a fragment of the syntax available in [1]. In particular we use parallel updates only in the atomic updates, while control state ASMs allow arbitrary ASM rules. It would be possible to generalize the atomic steps to general ASM rules, however, this would complicate code generation. Also, symbolic execution rules would get significantly more complex (cf. Chapter 8 of [1]), since parallel rules may have clashes.

For the atomic semantics given in Def. 3 it is not difficult to show that it agrees with standard rule semantics of ASM rules, when $\alpha; \beta$ is interpreted as α **seq** β in the following sense: $(s, s') \in [\![\mathtt{Op}]\!]$ corresponds to a successful computation of a consistent set of updates of a Turbo ASM rule in [1], Chapter 4 (and s' is the new state from applying this set). $(s, \perp) \in [\![\mathtt{Op}]\!]$ corresponds to either a diverging computation of updates, or to the computation of an inconsistent set. Our definition of submachines is different from the one in [1], where a submachine is a subrule that may be called within a rule, similar to a call of a submachine operation here.

For sequential ASMs with regular upate-set semantics as contexts, it should be possible to prove a modularity result comparable to the one Sec. 5 when submachine operations are represented by named procedures. Synchronously parallel submachine calls ($\mathtt{Op}^{\mathcal{L}}$ **par** $\mathtt{Op}^{\mathcal{L}}$) are problematic, because it would not be possible to extract a submachine run according to Prop. 1 and Def. 4. In practice, such parallel calls are also likely to produce clashes.

The crash theory of Sec. 6 could in principle be represented by control state ASMs, however, this introduces undesired overhead (see e.g. [24]).

Event-B has two decomposition concepts for machines that roughly correspond to interleaved [25] and synchronous parallel execution of rules [26]. It is not immediately clear how our submachine concept could be encoded by such a decomposition, since events in Event-B have no internal control structure.

8.2. File Systems & Crash-safety

The interrupt operator of CSP is similar to our semantic definition of a crash regarding the possibility to abort a running operation: Process $P \triangle_i Q$ denotes that P executes until an (external) event i occurs, after which P is discarded and Q is started. The operator originates from [27] and is further explored in [28]. In [29], a trace-based semantics is given that is equivalent to our white-box semantics of rules $[\![\alpha]\!]^{\square}$.

In [30] a high-level modeling language specifically designed for file systems is described. Examples cover sophisticated optimizations such as reordering of writes and versioning. However, the modeling language is not intended for an actual implementation of a file system. Bornholt et al. [31] similarly consider different crash consistency models and check the guarantees made by existing file systems in practice using the FERRITE tool.

In the context of operating system kernels, interrupts are similar to crashes in that they can occur at any time. The approaches we are aware of ([32, 33]) allow interrupts at specific locations in the code only.

In the context of the flash verification challenge, we know of several approaches that handle crashes. Kang et al. [34] uses the bounded model-checker Alloy [35] to analyze crashes during a write operation. However, the approach intertwines the effect of power loss with the specification of the write operation. The adequacy of the model is therefore hard to judge. The approach [36] uses bounded model-checking with SPIN to examine every intermediate state. Both approaches currently scale only to models that are specifically tuned for a small state space. Damchoom et al. [26] decompose the write operation at the granularity of pages using Event-B refinement. Crash safety is specified with respect to a shadow copy of the complete state (which is not realistic for Flash). The authors of [37] have performed an extensive analysis of existing file systems, including recovery, and have found multiple bugs.

Ridge et al. [38] provide a tool called SibylFS that serves as a test oracle and reference specification for existing file systems. Our implementation successfully passes the relevant tests.

Marić and Sprenger [39] consider a storage system which has similar properties as a file system, but with a strong focus on redundancy. They model crashes with exceptions that are thrown by hardware operations, which resembles our black-box machine $HW^{\blacksquare}$.

Chen et al. [40] discuss different formalisms to express crash and recovery on a high level. The follow up work [41] introduces Crash Hoare Logic in more detail and presents the verification of a small but complete file system called FSCQ targeting conventional magnetic drives. In comparison, their approach requires one to reason about all intermediate states using a special logic, whereas we are able to reduce the proof effort on a semantic level.

The approach in [42] annotates possibly nested regions of a program with a corresponding recovery operation. It is assumed that the runtime resp. the operating system knows which recoveries to start after a crash if this flexibility is used (in comparison of one top-level recovery in our approach). Modular abstraction of crashes and recovery is not supported by the formalism.

In both [41] and [42] each step of the program causes an additional proof obligation connecting a crash in the current state to the precondition of recovery.

Recent work by Koskinen and Yang [43] explores a strategy similar to our notion of crash neutrality: *Recovered programs should not introduce behaviors that were not present in the original (uncrashed) program* (Section 3.1). This amounts to the assumption that the original program is already functionally correct in the absence of crashes and that crashes are in fact completely unobservable (i.e., the abstract crash + recovery of our setting is just identity, which is insufficient for us). Supported by an automatic tool ELEVEN82, the authors have uncovered erroneous behavior in production quality data base systems.

We are aware of another ongoing efforts to construct a verified file system:

the authors in [44] have focused on a promising approach to simplify the transition to C-code using domain specific languages [45]. A model similar to our AFS is presented in [46] including a high-level functional specification the operation `fsync` to flush caches. Specification and verification methodology for crash safety is not described.

For an overview of existing efforts in the context of file system verification, see for example [47]. An extensive comparison of these efforts on a technical level to our approach can be found in [16, 12, 15].

9. Conclusion

We have defined a refinement theory for data type ASMs with submachines, which respect information hiding, and recovery from power failures. The theory has been a key to enable modular and incremental development of the flash file system case study.

In particular the reduction from crashes in intermediate to final states is an important contribution to the question of how to handle crashes with reasonable effort, that we found indispensable in practice: Usually, the most difficult proof on each level in the refinement hierarchy is showing a refinement between the concrete and abstract recovery operation. Considering the recovery explicitly at each step of a normal operation (and of the recovery itself) is virtually impossible.

Even with the theory we have presented here, developing a realistic flash file system, which has to bridge a large gap between abstract directory trees and low-level arrays to model flash pages is still a large undertaking, and we were only able to scratch the surface of the problems inherent in such a development in this paper. The specifications size is approximately 15k lines. Approximately half of the specifications are the rules of 19 ASMs, the other half are algebraic definitions. We generate 13k lines of C code as the final implementation.

The fact that we were neither limited to verifying low-level C code, nor to purely algebraic specifications of transition systems, has helped enormously to specify and verify on the right level of abstraction.

One key difficulty was integrating the solutions for the various aspects that have to be solved. Integrating the solutions properly was significantly more complex, than solving individual aspects in isolation like caching, orphans, dividing writes into pages etc. Often the integration of one additional aspect has lead to various modifications that propagated through several layers of refinements.

In particular integrating power cut safety is still one of the most difficult aspects. We estimate that at least half of the overall project of approximately three to four person years can be attributed to errors and power cut safety.

There are two important aspects, which we will consider in future work. Both will not cause tremendous changes to the final code, but amount to extensions and modifications that will affect the whole refinement tower in a non-local way.

First, the actual implementation uses concurrency to do work, such as garbage collection of erase blocks, in the background. On the syntactic level our approach is currently limited to consider sequential constructs only. We have been careful to define the syntax and semantics of our rules such that it is compatible with an extension to concurrency. In particular, our theorem prover KIV already supports a logic called RGITL (rely-guarantee interval temporal logic) for such an extension to interleaved programs [48]. Second, our models currently flush internal caches at the end of POSIX operations and therefore the effects of the operation can not be lost in a crash after the operation completed. The implementations in Linux, however, only flush when `fflush` or `sync` is called and it is therefore possible that a crash reverses several of the last operations.

References

[1] E. Börger, R. F. Stärk, Abstract State Machines — A Method for High-Level System Design and Analysis, Springer, 2003.

[2] E. Börger, The ASM Refinement Method, Formal Aspects of Computing 15 (1–2) (2003) 237–257.

[3] R. Joshi, G. Holzmann, A mini challenge: build a verifiable filesystem, Formal Aspects of Computing 19 (2).

[4] G. Reeves, T. Neilson, The Mars Rover Spirit FLASH anomaly, in: Aerospace Conference, IEEE Computer Society, 2005, pp. 4186–4199.

[5] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, W. Reif, Development of a Verified Flash File System, in: Proc. of ABZ 2014, Vol. 8477 of LNCS, Springer, 2014, pp. 9–24, (invited paper).

[6] G. Ernst, J. Pfähler, G. Schellhorn, Web presentation of the Flash Filesystem, `https://swt.informatik.uni-augsburg.de/swt/projects/flash.html` (2015).

[7] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, W. Reif, KIV - Overview and VerifyThis Competition, Software Tools for Techn. Transfer 17 (6) (2015) 677–694.

[8] J. C. P. Woodcock, J. Davies, Using Z: Specification, Proof and Refinement, Prentice Hall International Series in Computer Science, 1996.

[9] G. Ernst, J. Pfähler, G. Schellhorn, W. Reif, Modular Refinement for Submachines of ASMs, in: Proc. of ABZ 2014, Vol. 8477 of LNCS, Springer, 2014, pp. 188–203.

[10] The Open Group, The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2008 Edition, `http://www.unix.org/version3/online.html` (login required).

[11] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, W. Reif, A Formal Model of a Virtual Filesystem Switch, in: Proc. of Software and Systems Modeling (SSV), EPTCS, 2012, pp. 33–45.

[12] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, W. Reif, Verification of a Virtual Filesystem Switch, in: Proc. of Verified Software: Theories, Tools, Experiments (VSTTE), Vol. 8164 of LNCS, Springer, 2014, pp. 242–261.

[13] D. Harel, D. Kozen, J. Tiuryn, Dynamic Logic, MIT Press, 2000.

[14] W. Reif, G. Schellhorn, K. Stenzel, M. Balser, Structured specifications and interactive proofs with KIV, in: W. Bibel, P. Schmitt (Eds.), Automated Deduction—A Basis for Applications, Vol. II, Kluwer, Dordrecht, 1998, pp. 13–39.

[15] J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, W. Reif, Formal Specification of an Erase Block Management Layer for Flash Memory, in: Hardware and Software: Verification and Testing, Vol. 8244 of LNCS, Springer, 2013, pp. 214–229.

[16] G. Ernst, J. Pfähler, G. Schellhorn, W. Reif, Inside a Verified Flash File System: Transactions & Garbage Collection, in: Proc. of Verified Software: Theories, Tools, Experiments (VSTTE), Vol. 9593 of LNCS, Springer, 2016, pp. 73–93.

[17] J. He, C. A. R. Hoare, J. W. Sanders, Data refinement refined, in: Proc. of the European symposium on programming on ESOP 86, Springer-Verlag New York, Inc., 1986, pp. 187–196.

[18] G. Schellhorn, ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison, Journal of Theoretical Computer Science 336 (2–3) (2005) 403–435.

[19] W. de Roever, K. Engelhardt, Data Refinement: Model-Oriented Proof Methods and their Comparison, Vol. 47 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[20] G. Schellhorn, Verification of ASM Refinements Using Generalized Forward Simulation, Journal of Universal Computer Science (J.UCS) 7 (11) (2001) 952–979, URL: `http://www.jucs.org`.

[21] G. Schellhorn, Completeness of Fair ASM Refinement, Science of Computer Programming, Elsevier 76, issue 9 (2009) 756 – 773.

[22] J.-R. Abrial, Modeling in Event-B, Cambridge University Press, 2010.

[23] J.-R. Abrial, The B Book - Assigning Programs to Meanings, Cambridge University Press, 1996.

[24] E. Börger, J. Schmid, Composition and Submachine Concepts for Sequential ASMs, in: P. Clote, H. Schwichtenberg (Eds.), Proc. 14th International Workshop Computer Science Logic (Gurevich Festschrift), LNCS 1862, Springer, 2000, pp. 41–60.

[25] J.-R. Abrial, S. Hallerstede, Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B, Fundamenta Informaticae 77.

[26] K. Damchoom, M. Butler, Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B, in: Formal Methods: Foundations and Applications, Springer, 2009, pp. 134–152.

[27] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[28] A. A. McEwan, J. Woodcock, Unifying theories of interrupts, in: A. Butterfield (Ed.), Unifying Theories of Programming, Vol. 5713 of Lecture Notes in Computer Science, Springer, 2010, pp. 122–141.

[29] A. W. Roscoe, C. A. R. Hoare, R. Bird, The Theory and Practice of Concurrency, Prentice Hall, 1997.

[30] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, S. Jha, A logic of file systems., in: FAST, Vol. 5, 2005, pp. 1–1.

[31] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, X. Wang, Specifying and checking file system crash-consistency models, in: Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, 2016, pp. 83–98.

[32] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, seL4: Formal Verification of an OS Kernel, in: Proc. of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, ACM, New York, NY, USA, 2009, pp. 207–220.

[33] C. Baumann, T. Bormer, Verifying the PikeOS Microkernel: An Overview of the Verisoft XT Avionics Project, in: 4th International Workshop on Systems Software Verification (SSV 2009), Electronic Notes in Theoretical Computer Science, Elsevier Science B.V., 2009, (to appear).

[34] E. Kang, D. Jackson, Designing and Analyzing a Flash File System with Alloy, Int. J. Software and Informatics 3 (2-3) (2009) 129–148.

[35] D. Jackson, Software Abstractions: Logic, Language, and Analysis, MIT Press, 2006.

[36] P. Taverne, C. Pronk, RAFFS: Model Checking a Robust Abstract Flash File Store, in: Proc. of Int. Conf. on Formal Engineering Methods: Formal Methods and Software Engineering, ICFEM '09, Springer, 2009, pp. 226–245.

[37] J. Yang, P. Twohey, D. Engler, M. Musuvathi, Using Model Checking to Find Serious File System Errors, in: Proc. of OSDI, USENIX, 2004, pp. 273–288.

[38] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, P. Sewell, SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems, in: Proc. of SOSP, ACM, 2015.

[39] O. Marić, C. Sprenger, Verification of a transactional memory manager under hardware failures and restarts, in: FM 2014: Formal Methods, Springer LNCS 8442, 2014, pp. 449–464.

[40] H. C., D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, N. Zeldovich, Specifying crash safety for storage systems, in: 15th Workshop on Hot Topics in Operating Systems (HotOS XV), USENIX Association, 2015.

[41] H. Chen, D. Ziegler, A. Chlipala, N. Zeldovich, M. F. Kaashoek, Using crash hoare logic for certifying the FSCQ file system, in: Proc. of SOSP, ACM, 2015, pp. 18–37.

[42] G. Ntzik, P. da Rocha Pinto, P. Gardner, Fault-tolerant resource reasoning, in: Proc. of the Asian Symposium on Programming Languages and Systems (APLAS), Springer LNCS 9458, 2015, pp. 169–188.

[43] E. Koskinen, J. Yang, Reducing crash recoverability to reachability, in: Proc. of Principles of Programming Languages (POPL), ACM, 2016.

[44] G. Keller, T. Murray, S. Amani, L. O'Connor, Z. Chen, L. Ryzhyk, G. Klein, G. Heiser, File systems deserve verification too!, in: Proc. of the 7th Workshop on Programming Languages and Operating Systems, PLOS, ACM, 2013, pp. 1–7.

[45] L. O'Connor-Davis, G. Keller, S. Amani, T. Murray, G. Klein, Z. Chen, C. Rizkallah, CDSL version 1: Simplifying verification with linear types, Tech. rep., NICTA (2014).

[46] S. Amani, T. Murray, Specifying a realistic file system, in: Workshop on Models for Formal Analysis of Real Systems, 2015, pp. 1–9.

[47] M. Lali, File system formalization: Revisited, Int. Journal of Advanced Computer Science 3 (12) (2013) 602–606.

[48] G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, W. Reif, RGITL: A temporal logic framework for compositional reasoning about interleaved programs, Annals of Mathematics and Artificial Intelligence (AMAI) 71 (2014) 131–174.

List of Symbols

Symbol	Description	Page
f	A function symbol	5
p	A predicate symbol	5
SIG	A signature	5
x	A variable	5
$\underline{x}$	A tuple (of variables)	5
t	A term	5
φ	A formula	5
ε	A quantifier-free formula	5
a	An element of an algebra	5
$s \in S$	A state (= algebra and valuation of variables)	5
$s\{x \mapsto a\}$	A state with modified variable	5
$s\{f(\underline{t}) \mapsto a\}$	A state with updated function	5
loc	A location (either $f(\underline{t})$ or x)	5
$[\![t]\!](s)$	The semantics of a term	5
α, β	A program	5
I	An interval (poss. inf. sequence of states)	6
$\#I$	The number of steps of an interval	6
$I\{\underline{x} \mapsto \underline{\boldsymbol{a}}\}$	Interval with modified variables	6
$I_1 seq I_2$	Sequential composition of intervals	6
$(s, \ldots, s')$	Interval with finite length	6
$(s, \ldots)$	Interval with infinite length	6
$I \models \alpha$	Nonatomic semantics of a program	6
$S_\perp$	States with $\perp$	7
$[\![\alpha]\!] \subseteq S_\perp \times S_\perp$	Atomic semantics of a program	7
$\langle\!|\alpha|\!\rangle\, \varphi$	α always terminates and φ holds at the end	7
$\langle\alpha\rangle\, \varphi$	α has a terminating run with φ	7
$\mathcal{M}$	Data type ASM	8
pre	A precondition	8
$\underline{in}$	A tuple of input locations	8
$\underline{out}$	A tuple of output locations	8
$\mathtt{Op} = (pre, \underline{in}, \alpha, \underline{out})$	An operation	8
$j \in J$	Indexes of operations of a data type ASM	8
$\mathtt{Op}_j$	An operation of a data type ASM	8
$[\![\mathtt{Op}]\!] \subseteq S_\perp \times S_\perp$	Atomic semantics of an operation	8
$\underline{j} = (j_0, j_1, \ldots)$	finite or infinite call sequence	9
$exec^{\mathcal{M}}(\underline{j})$	executions of call sequence $\underline{j}$ (a set of intervals)	9
$runs^{\mathcal{M}}(\underline{j})$	runs of call sequence $\underline{j}$ (a set of intervals)	9
$\mathcal{A}, \mathcal{C}$	Abstract and concrete machine	9
$as \in AS, cs \in CS$	Abstract and concrete states	9
$\mathtt{Op}_j^{\mathcal{A}}, \mathtt{Op}_j^{\mathcal{C}}$	Abstract and concrete operation	9
$IO \subseteq AS \times CS$	Input/Output correspondence	9
$I^{\mathcal{C}} \sqsubseteq_{IO} I^{\mathcal{A}}$	Interval $I^{\mathcal{C}}$ refines interval $I^{\mathcal{A}}$ with IO	9

Symbol	**Description**	**Page**
$\mathcal{C} \sqsubseteq_{IO} \mathcal{A}$	$\mathcal{C}$ refines $\mathcal{A}$	9
$R \subseteq AS \times CS$	A forward simulation	9
$R_1 seq R_2$	Relational composition of two relations	9
$Init^{\mathcal{A}} \lhd R$	Domain restriction of relation R to set $Init^{\mathcal{A}}$	9
$ran(R)$	Range of relation R ($\subseteq CS$)	9
$\mathcal{L}, \mathcal{K})$	Submachines (Data type ASMs)	10
ls,ks	States of machine $\mathcal{L}$ and $\mathcal{K}$	10
$\mathcal{M}(\mathcal{K})$	Machine $\mathcal{M}$ (a data type ASM) which uses machine $\mathcal{K}$	10
$ls{\oplus}gs$	A states of machine $\mathcal{M}(\mathcal{L})$	10
$ks{\oplus}gs$	A states of machine $\mathcal{M}(\mathcal{K})$	10
$I, \underline{c} \models \alpha$	Nonatomic semantics of a program with explicit submachine call sequence	11
$exec^{\mathcal{L}}_{\tau}(\underline{k})$	Stuttering execution with τ steps	12
$\mathcal{M}^{\square}$	Machine $\mathcal{M}$ with crashes at any time ("white-box")	16
$\mathcal{M}^{\blacksquare}$	$\mathcal{M}$ with crashes only between operations ("black-box")	16
$\mathcal{M}^{\text{⬔}}$	Extended machine which can have crashes	16, 18
$\mathcal{M}^{\square}(\mathcal{L}^{\square})$	Crashes only between $\mathcal{M}$-operations	16
$\mathcal{M}^{\blacksquare}(\mathcal{L}^{\square})$	Crashes within $\mathcal{M}$ ops., but with atomic $\mathcal{L}$ ops	16
$\mathcal{M}^{\blacksquare}(\mathcal{L}^{\blacksquare})$	Crashes even within $\mathcal{L}$-operations	16
$\mathtt{Op}_{\mathrm{rec}}$	Distinguished operation called for recovery from a crash	18
$Cr \subseteq S \times S$	Relation describing the behavior of a crash	18
$s_{\lightning} \in S_{\lightning}$	state with crash marker	18
$[\![\alpha]\!]^{\blacksquare}$	White-Box semantics of a rule with crashes	18
$[\![\alpha]\!]^{\blacksquare}$	Black-Box semantics of a rule with crashes	19
$[\![\mathtt{Op}_j]\!]^{\text{⬔}} \subseteq S_{\perp} \times (S_{\perp} \uplus S_{\lightning})$	Black- and white-box semantics of an operation	19