Modular Refinement for Submachines of ASMs *

Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif

Institute for Software & Systems Engineering
University of Augsburg, Germany
{ernst,joerg.pfaehler,schellhorn,reif}
@informatik.uni-augsburg.de

Abstract. We describe and formalize a compositional, contract-based submachine refinement for a variant of Abstract State Machines. We motivate the approach by models of the Flash file system case study, where it is infeasible to refine a complete machine as a whole.

1 Introduction

Abstract State Machines (ASMs, [5]) are a general software development method for state-based systems. By integrating them with refinement [4] and the algebraic specification of data types they provide a rigorous framework for verifying correctness critical applications.

This paper contributes a formally defined instance of ASM refinement theory, which is *compositional* for submachines that respect information hiding. Formally, we prove that a machine $\mathcal{M}$ that calls the operations of a submachine $\mathcal{L}$ satisfies the following substitution law: If $\mathcal{K}$ is a correct refinement of $\mathcal{L}$, then substituting calls to $\mathcal{L}$ with calls to $\mathcal{K}$ in $\mathcal{M}$ is a refinement of $\mathcal{M}$. The theorem allows to refine submachines independently of the context formed by $\mathcal{M}$.

This work is strongly motivated by our current effort to construct a verified file system for flash memory. This challenge has been proposed by NASA [15] in response to problems with the flash file system of the Mars Rover "Spirit" [20].

As a consequence, the syntax and semantics of the variant of ASMs we consider here differs and is somewhat restricted compared to traditional ASMs. In particular, we define both an *atomic semantics* and an *non-atomic semantics* for the rules. The former is intended for the environment of rules, e.g. the caller of a POSIX operation like "create directory". It is also the semantics of calls to submachines. The non-atomic semantics is necessary to study the effects of power failures while an operation (i.e., a rule of a control-state ASM which goes through a potentially infinite sequence of intermediate states) is running: the recovery mechanism that runs when rebooting after a power failure must restore a consistent state from any intermediate state.

* The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-662-43652-3_16

On the syntactic level our approach currently considers sequential constructs only, since we have not investigated concurrent execution for the Flash file system. Thus, the atomic semantics is an instance of sequential ASMs, the non-atomic semantics is an instance of control-state ASMs with a single control state. An extension to several control states and interleaved execution is possible, at the price that reasoning with the wp-calculus has to be replaced with the more complex reasoning using the temporal logic RGITL we define in [**?**]. The ASM rules we use here could also be called (a subset of) RGITL programs.

Our earlier work on ASM refinement [22,24] assumed that rules have *guards*, and that an ASM chooses to invoke rules only when the guard is true. This view is not suitable for submachines, where no guarantee can be given that calls to the submachine will respect the guards, nor that a refined submachine (viewed in isolation) has fewer runs than the original submachine with the guard view. Instead we will use *preconditions* for rules, where a call with precondition false is possible, but results in arbitrary behavior. Our approach takes up ideas from *contract refinement* as used in Z [28], and adapts it to our scenario.

We motivate the need for compositional refinement of submachines in Sec. 2, by giving an overview over the structure of our development. Sec. 3 demonstrates why preconditions are necessary instead of guards. For reasons of space, we only sketch the machines needed. For an overview over the project we refer the reader to [25] (this volume). The full details and a list of previous publications is available online [9]. Sec. 4 defines syntax and semantics of the ASM rules we use. We contribute two compatible semantic definitions: a *non-atomic* view, where execution of an ASM rule results in a sequence of steps; and an *atomic* view, on which we base the definition of runs of an ASM, the semantics of submachine calls, and the weakest precondition calculus we use for deduction in our prover KIV [21]. Sec. 5 defines refinement for our setting, gives the proof obligations for forward simulation, and proves that refinement is modular for submachines. Sec. 6 gives related work and Sec. 7 concludes.

2 Submachines in the Flash File System

In this section, we briefly show the topmost refinement of the refinement hierarchy and motivate that the file system challenge is inherently compositional. Figure 2 displays the structure of the topmost part of the project, where boxes represent components, and layers respectively, connected by refinement (dotted lines). These are formally given by Abstract State Machines (ASMs) with algebraic states. The grey boxes are the leaves of the hierarchy from which we will generate the final code.

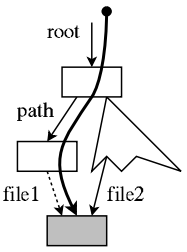

Fig. 1. FS graph

At the toplevel, POSIX [27] specifies the requirements: The file system (FS) is a graph consisting of directories (internal nodes) and files (leaves), an example is shown in Fig. 1. Files can be referred to by multiple directories under different names ("hard-links"), consequently, names are attached

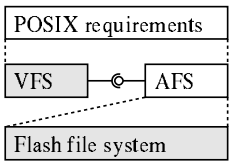

Fig. 2. FFS upper layers

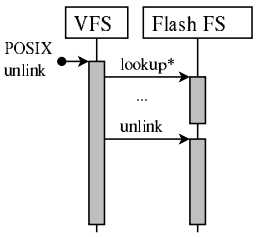

Fig. 3. Call sequence in the final composed file system code

```
posix_unlink(path; err)
  tree := tree − path
  // conditionally delete content
```

Fig. 4. POSIX specification of `unlink`

```
vfs_unlink(path; err)
  ino := ROOT_INO;
  while path.length > 1 do {
     let n = path.head
     in afs_lookup(ino, n; ino′, err);
     path := path.tail, ino := ino′
  }
  let dent = negdentry(path.head)
  in afs_unlink(ino; dent, err)
  // conditionally evict ino

pre: dirs[ino] ≠ undef ∧ ...
afs_unlink(ino; dent, err)
  let name = dent.name in
  dirs[ino].entries[name] := undef
```

Fig. 5. VFS/AFS rules (no error-handling, submachine calls underlined)

to edges of the graph. The directory part is a proper tree. The POSIX interface is based on *paths*. Our formal POSIX model can be found in [11]. As an example, Fig. 4 shows the specification of the `unlink` operation, which removes one link to the file denoted by *path*, and also deletes the file's content once it is unreferenced.

Real file system implementations consist of two parts. Generic aspects, i.e. traversing paths and checking access rights are realized in Linux by the *Virtual Filesystem Switch* (VFS). Windows' "Installable File System" serves the same purpose. Concepts specific to individual file system implementations are realized by the individual file systems (FS). For standard magnetic discs ext4 or ReiserFS would be such file systems, for flash memory we use UBIFS [14] as a design blueprint for our formal models.

VFS communicates with individual file systems through a well-defined interface visualized by the symbol ⊸◎⊸ in Fig. 2. The main data structure used in this interface is called *inodes* in Linux. To specify this interface we define an ASM called *Abstract File System* (AFS). Technically, AFS is a *submachine* of VFS.

A typical ASM rule for the VFS operation `unlink` is sketched in Fig. 5 (full models can be found in [10]). Several calls of `afs_lookup` are used to traverse the path, checking that the individual directories exist with suitable access rights. Finally, `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. The use of an abstract AFS specification makes the proof that VFS is a proper refinement of the POSIX specification [11] *independent* of the actual file system.

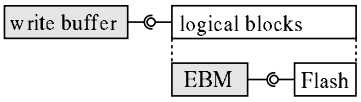

Fig. 6. FFS lower layers

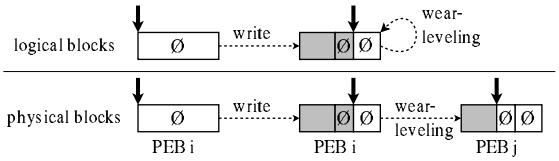

Fig. 7. Abstract/concrete run with wear-leveling

Figure 3 shows the corresponding sequence of operations in the final *composed code* we generate (marked grey in Fig. 2). In this code calls to abstract AFS operations have been replaced by calling the concrete FS code. The main theorem we prove in this paper, is that this methodology is sound, i.e., the composed code refines the top-level specification (see Theorem 2 in Sec. 5).

As expected, the size of implementation components is much larger than the size of their respective specifications (POSIX: 50 lines, VFS: 500 lines, AFS: 100 lines, abstract specification of UBIFS again calling some submachines: 500 lines). This pattern repeats all the way down to the hardware interface, forming a *deep* hierarchy. An approach that exploits the compositional structure is key to make the verification of the whole file system tractable.

3 Contracts in the Eraseblock Management Layer

This section motivates why we need preconditions, not guards for our refinements. In the following we present a (simplified) example taken from the lower layers of our flash file system and show how this technique is employed to simplify the abstract layer.

Figure 6 shows a section of these layers. We have a model of flash hardware at the bottom. A device is divided into *physical erase blocks* (PEBs). The basic limitation of flash hardware is that only *sequential* writes within a block are allowed and overwriting is not possible. Space can only be reclaimed by *erasing* a full block. Erasing is slow and physically degrades the memory, i.e., after 10^4-10^6 erasures a block is unusable. In order to increase the reliability and lifetime of the device, the layer directly above the hardware, the Erase Block Management (EBM), performs *wear-leveling* in the background. Wear-leveling moves stale data to new blocks, in order to spread the erases evenly amongst the blocks. This is implemented transparently by introducing a mapping from *logical* to *physical* erase blocks. The mapping is managed by the EBM internally, the client can only access logical erase blocks (LEBs). We then abstract this implementation to a layer similar to the flash device ("logical blocks" in the figure). The aim of the following is to show how preconditions can be used to hide wear-leveling from upper layers. For technical details not covered here, the reader is referred to [18].

The basic idea to specify the limitation to sequential writes on physical and their abstraction to logical blocks is to associate an offset called *fillcount* with

each block. It stores how far the block is already programmed with data. This offset can not be accessed by the EBM and is only used to express the precondition of a write (to logical as well as physical blocks): the offset of the write is above the *fillcount* of the target block. The vertical arrows ↓ in Fig. 7 denote the *fillcount*. The upper part of the figure shows a logical block and how it is affected by a write operation and a subsequent wear-leveling cycle. The lower half depicts the PEB mapped for this LEB at any point in time. During writing the *fillcount* fields are affected in the same way. For efficiency the implementation of wear-leveling, however, does not copy the entire contents of the block (if the remainder is already empty). This means that in general the target block j may have a lower *fillcount* than the source block i, as shown in the figure.

If the guard-semantics is used, the EBM model has more runs than its abstraction: In Fig. 7 for example after wear-leveling additional writes to physical block j are possible, which the abstract model can not reproduce. These additional runs, however, will never be exploited by a client, since their existence crucially depends on wear-leveling, which is not triggered by the client but performed non-deterministically in the background.

In conclusion, preconditions are more appropriate for submachines than the guards. The reason is that operations of a submachine are explicitly called and are not triggered internally. Therefore, an implementation may have a more liberal precondition than the abstract system. This can be used to simplify the abstract system by strengthening the abstract precondition and thereby hiding inconsequential runs of the concrete system. Refinement can still be expressed as the usual trace inclusion if all possible runs (including divergence) are added to the semantics of an operation outside of its precondition (see Def. 3 and 5).

4 Syntax and Semantics of ASMs with Submachines

Section 4.1 defines the syntax of ASMs without submachines. Roughly, an ASM consists of a number of rules with preconditions (called "operations"). The rules are given a non-atomic semantics in Sec. 4.2 that is similar to the one of control state ASMs, however, we never use an explicit control state. We then abstract to an atomic view, which is used to define runs of ASMs in Sec. 4.3 and calls to submachine operations in Sec. 4.4. Finally, Sec. 4.5 gives the semantics of wp-formulas that we use to define and verify properties.

4.1 Syntax

This subsection defines the syntax of the ASMs we use. 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 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 ASM rules), an *output* signature (that is only written by ASM rules), and a *controlled* signature that may be read and written by ASM rules.

We use the general convention to underline sequences of elements, i.e., $\underline{a}$ stands for a sequence $(a_1, \ldots, a_n)$ 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 (so no clashes are possible) and writable, i.e., are local variables or part of the controlled or output signature. We write **skip** for an empty parallel update.

The **choose** constructs 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

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

where $\underline{y}$ are new variables and $\alpha^{\underline{y}}_{\underline{x}}$ denotes the substitution of $\underline{x}$ with $\underline{y}$ in α.[1] Note that **ifnone skip** is never executed here, and we will we drop such irrelevant ifnone-clauses as well as random choice (i.e., **with true**) in the following.

Based on the syntax of rules we define abstract state machines.

Definition 1. *A (data type-like) ASM* $\mathcal{M} = (SIG, Ax, Init, \{\texttt{Op}_j\}_{j \in J})$ *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* $\texttt{Op}_j = (pre_j, \underline{in}_j, \alpha_j, \underline{out}_j)$ *consists of an ASM rule* α_j *that describes possible state transitions, provided precondition* pre_j *holds. It reads input from a vector* $\underline{in}_j$ *of input locations, and writes output to a vector* $\underline{out}_j$ *of output locations. It may modify local variables, controlled locations and the locations of* $\underline{out}_j$*. The rules should have no non-local variables.*[2]

In concrete code like the one given in Fig. 4 each operation $\texttt{Op}_j$ has a *name* (instead of using an index j), the precondition is given after keyword **pre**, and the other components are given in the form of $name(\underline{in}_j; \underline{out}_j)\{\ \alpha_j\ \}$.

[1] The renaming avoids conflicts when $\underline{x}$ is used in $\underline{t}$.

[2] Thus, states of $\mathcal{M}$ are just *SIG*-Algebras; the values of variables are irrelevant.

4.2 Non-atomic Semantics of Rules

This section gives a semantics to rules that assumes that they are executed non-atomically: each update and each test of a condition is executed as a separate step. The semantics of rules is therefore based on sequences $I = (I(0), I(1), \ldots)$ of states $I(k)$, which may be finite or infinite. Such sequences are called *intervals*. Formally, $I \models \alpha$ expresses that the interval I is a possible execution of α.

We introduce some auxiliary notation: The length of an interval $\#I$ 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 also lift modification of states to intervals: given a vector of variables $\underline{x}$ and a sequence of value vectors $\underline{\boldsymbol{a}} = (\underline{a}_0, \underline{a}_1, \ldots)$ of the same length as the interval (where each element $\underline{a}_k$ has the length of vector $\underline{x}$), then $I\{\underline{x} \mapsto \underline{\boldsymbol{a}}\}$ is the modified interval, where $I(k)(\underline{x})$ is $\underline{a}_k$.

For sequential execution we need the sequential composition of intervals I_1 and I_2, written $I_1 \fatsemi I_2$, which is defined in two cases. For finite I_1, the last state of I_1 (written $I_1.\mathrm{last}$) must agree with the first of I_2: $I_1.\mathrm{last} = I_2(0)$, and the result is $(I_1(0), \ldots, I_1.\mathrm{last}, I_2(1), I_2(2), \ldots)$, i.e., the duplicate middle state is removed. If I_1 is infinite, then $I_1 \fatsemi I_2 := I_1$.

Definition 2.

$I \models \underline{loc} := \underline{t}$ 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 $I = I_1 \fatsemi I_2$

$I \models$ **if** ε **then** α **else** β
iff either $I(0) \models \varepsilon$ and $I \models$ **skip**$; \alpha$ or $I(0) \not\models \varepsilon$ and $I \models$ **skip**$; \beta$

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

$I \models$ **while** ε **do** α
iff $I \in \nu(\lambda\, \mathcal{I}.\ \{I_0 \mid$
either $I_0(0) \not\models \varepsilon$ and $I_0 \models$ **skip**
or $\#I_0 = \infty$ and $I_0(0) \models \varepsilon, I_0 \models$ **skip**$; \alpha$
or $I_0 = I_1 \fatsemi I_2$ with $\#I_1 < \infty, I_1(0) \models \varepsilon, I_1 \models$ **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.[3] In the first disjunct of the semantics of choose, the sequence of states $\underline{\boldsymbol{a}}$ captures the values $\underline{x}$ 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[4] of interval sets $\mathcal{I}$ whose elements

[3] It is possible (and sometimes useful; e.g., to define atomic test-and-set instructions), to define the semantics such that evaluations of tests takes no additional step.

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

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 case $I_1(0) \models \varepsilon, I_1 \models \mathbf{skip}; \alpha$), 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 case of the definition), or it may be one **skip** step, where the loop test evaluates to false (first case of the definition).

4.3 Semantics of ASMs

The interval semantics of rules is relevant when we want to study the effect of power cuts, which will interrupt execution and produce prefixes of the intervals. It is also relevant when we add calls to submachines in the next subsection.

To define the runs of a machine $\mathcal{M}$, the fine-grained semantics of rules can be abstracted to an atomic view of the execution of an operation. It is also much easier to reason about the atomic semantics (using wp-calculus, see Sec. 4.5). The atomic view is therefore a relation over the set of states augmented with a bottom element, to indicate nontermination: $S_\perp := S \cup \{\perp\}$.

Definition 3. *The atomic semantics* $[\![\mathtt{Op}]\!] \subseteq S_\perp \times S_\perp$ *of an ASM operation* $\mathtt{Op} = (pre, \underline{in}, \alpha, \underline{out})$ *is defined as:*

$$
\begin{aligned}
(s, s') \in [\![\mathtt{Op}]\!] \quad \text{iff either } & s \neq \perp \text{ and } s \not\models pre \quad (\text{and } s' \text{ is arbitrary from } S_\perp) \\
\text{or } & s \neq \perp, s \models pre \text{ and there is } I \text{ with } I(0) = s, I \models \alpha \\
& \quad \text{and if } I \text{ is finite then } s' = I.\text{last}, \text{ otherwise } s' = \perp \\
\text{or } & s = s' = \perp
\end{aligned}
$$

The first line of the definition gives the idea of a precondition: if it is violated in state s, then calls to α may result in any successor state (including nontermination). The second clause collapses terminating runs I of α to their first and last state. Infinite runs yield $\perp$. 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 $\perp$.

Based on the semantics of single operations we can define runs of a machine:

Definition 4. *An ASM program over a machine* $\mathcal{M}$ *is a possibly infinite sequence* $\underline{j} = (j_0, j_1, \ldots)$ *of (indices or names of) operation calls. An execution of the program* $\underline{j}$ *is an interval* $\boldsymbol{I}$ *with states in* $S_\perp$ *and* $\#\boldsymbol{I} = \#\underline{j}$,[5] *where*

$$(s, s') \in [\![\mathtt{Op}_{j_k}]\!] \quad \text{for } s = \boldsymbol{I}(k) \text{ and } s' = \boldsymbol{I}(k+1)\{\underline{in}_{j_k} \mapsto s(\underline{in}_{j_k})\}$$

holds for all $0 \leq k < \#\boldsymbol{I}$. *An execution is a run of the program, written* $\boldsymbol{I} \in runs^{\mathcal{M}}(\underline{j})$ *if it starts with an initial state* $\boldsymbol{I}(0) \neq \perp$, $\boldsymbol{I}(0) \models Init$.

[5] We write intervals that may contain $\perp$ in bold.

The definition of runs of programs mimics the definition of runs of data types, although we consider both finite and infinite runs. Note that state $\boldsymbol{I}(k)$ stores the input $s(\underline{in}_{j_k})$ for calling $\mathtt{Op}_{j_k}$. The operation itself does not change it, so state s' still stores the old input. Instead, the environment of the ASM is assumed to modify the input arbitrarily to the next one stored in $\boldsymbol{I}(k+1)$.

In contrast to the standard definition of guarded rules, runs as defined here 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, or execution may continue with an arbitrary state.

4.4 Submachines

Now we define ASMs $\mathcal{M} = (SIG, Ax, Init, \{\mathtt{Op}_j\}_{j\in J})$ that call operations of a submachine $\mathcal{L} = (SIG^{\mathcal{L}}, Ax^{\mathcal{L}}, Init^{\mathcal{L}}, \{\mathtt{Op}_k^{\mathcal{L}}\}_{k\in K})$. We require that $\mathcal{M}$ uses $\mathcal{L}$ *properly*, indicated by the notation $\mathcal{M}(\mathcal{L})$. The following conditions must be satisfied:

- $\mathcal{M}$ extends $\mathcal{L}$'s signature and axioms: $SIG^{\mathcal{L}} \subseteq SIG$ and $Ax^{\mathcal{L}} \subseteq Ax$,
- initialization of $\mathcal{M}$ includes initialization of $\mathcal{L}$, i.e., $Init \to Init^{\mathcal{L}}$ holds and
- $\mathcal{M}$ 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 or tests of $\mathcal{M}$ operations. We write ls for this local state (and similarly ls' for the local part of s'). The global state gs is the state of $\mathcal{M}$ without ls.

Rules α_j of the ASM may now contain calls to operations of $\mathcal{L}$. We extend the syntax of rules of Sec. 4.1 with

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

The call copies (values of) actual input parameter terms $\underline{t}$ to the input locations $\underline{in}_k^{\mathcal{L}}$ of $\mathtt{Op}_k^{\mathcal{L}}$, executes the rule $\mathtt{Op}_k^{\mathcal{L}}$, and finally copies $\underline{out}_k^{\mathcal{L}}$ back to actual outputs $\underline{loc}$, which must be writable locations of $\mathcal{M}$. Within the run of α_j the call to $\mathtt{Op}_k^{\mathcal{L}}$ is considered as one atomic step. The semantics is therefore defined as

$$\begin{aligned} I \models \mathtt{Op}_k^{\mathcal{L}}(\underline{t}; \underline{loc}) \\ \text{iff} \quad & I = (s, s'\{\underline{loc} \mapsto \underline{out}_k^{\mathcal{L}}\}) \text{ and } gs' = gs \text{ and } (ls\{\underline{in}_k^{\mathcal{L}} \mapsto \underline{t}\}, ls') \in [\![\mathtt{Op}_k^{\mathcal{L}}]\!] \\ \text{or } & \#I = \infty, I(0) = s \text{ and } (ls\{\underline{in}_k^{\mathcal{L}} \mapsto \underline{t}\}, \bot) \in [\![\mathtt{Op}_k^{\mathcal{L}}]\!] \end{aligned}$$

Note that we avoid adding $\bot$ to the non-atomic semantics here: if the call of $\mathtt{Op}_k^{\mathcal{L}}$ does not terminate, then the resulting interval for $\mathtt{Op}_k^{\mathcal{L}}$ is infinite, implying that the operation calling $\mathtt{Op}_k^{\mathcal{L}}$ does not terminate, too. Intervals $I \in runs^{\mathcal{M}}(\underline{j})$ therefore contain the outputs of the submachine (in the formal outputs $\underline{out}_k^{\mathcal{L}}$).

Given an interval $I \models \alpha$, where α calls operations of a submachine $\mathcal{L}$, it is possible to extract the execution $I\big|_{\mathcal{L}}$ of the submachine. It is an interval

over the state space of $\mathcal{L}$ including $\bot$ and its length matches the number of submachine calls in I. For every call (s, s') in I, $I\big|_{\mathcal{L}}$ has a state transition (ls, ls'), all other transitions are left out. If the last call to an operation of $\mathcal{L}$ starting in a state s does not terminate, a transition $(ls, \bot)$ is added. Note that merging the intervals of consecutive calls to $\mathcal{L}$ is possible because the local state is not altered in between the calls. We lift this definition to an execution $\boldsymbol{I}$ of a program $\underline{j}$ on $\mathcal{M}(\mathcal{L})$: $\boldsymbol{I}\big|_{\mathcal{L}}$ is the concatenation of the submachine intervals of each of the operations of $\underline{j}$. It follows:

Lemma 1. *Given an execution $\boldsymbol{I}$ of a program of $\mathcal{M}(\mathcal{L})$, then $\boldsymbol{I}\big|_{\mathcal{L}}$ is an execution of $\mathcal{L}$. It is a run if $\boldsymbol{I}$ is.* □

4.5 Calculus

To define and to verify properties of ASM rules we use the wp-calculus. The calculus defines two program formulas $\langle\!|\alpha|\!\rangle\, \varphi$ and $\langle \alpha \rangle\, \varphi$ as follows:

$$
\begin{aligned}
s \models \langle\!|\alpha|\!\rangle\, \varphi \quad & \text{iff all intervals } I \models \alpha \text{ with } I(0) = s \text{ have } \#I < \infty \text{ and } I.\mathrm{last} \models \varphi \\
s \models \langle \alpha \rangle\, \varphi \quad & \text{iff there is a finite interval } I \models \alpha \text{ with } I(0) = s \text{ and } I.\mathrm{last} \models \varphi
\end{aligned}
$$

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 [12] and expresses that α has a terminating run after which φ holds.[6]

Note that in contrast to standard wp-calculus formula φ is not restricted to a predicate logic formula, 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 [21]; an extension of symbolic execution to temporal logic formulas is described in [**?**]).

5 Refinement of ASMs and of Submachines

5.1 Contract Refinement for ASMs

ASM 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 ("input/output correspondence") over the input and output part of the two algebras. It specifies what matching inputs and outputs are. Often IO requires identity for input and output locations, but more general cases are possible. $IO(as, cs)$ is given syntactically as a formula over the combined signature $SIG^{\mathcal{A}} \dot\cup SIG^{\mathcal{C}}$. Correspondence of two executions

[6] Dynamic Logic writes $\mathbf{wlp}(\alpha, \varphi)$ as $[\alpha]\, \varphi$; $\langle \alpha \rangle\, \varphi$ is equivalent to $\neg\, [\alpha]\, \neg\, \varphi$.

$\boldsymbol{I}^{\mathcal{C}}$ and $\boldsymbol{I}^{\mathcal{A}}$ of $\mathcal{C}$ and $\mathcal{A}$ ("$\boldsymbol{I}^{\mathcal{C}}$ matches $\boldsymbol{I}^{\mathcal{A}}$ via IO") is defined as

$$\begin{aligned}\boldsymbol{I}^{\mathcal{C}} \sqsubseteq_{IO} \boldsymbol{I}^{\mathcal{A}} \quad & \text{iff } \#\boldsymbol{I}^{\mathcal{C}} = \#\boldsymbol{I}^{\mathcal{A}} \text{ and for all } k < \#\boldsymbol{I}^{\mathcal{C}} : \\ & \text{either } \boldsymbol{I}^{\mathcal{A}}(k) = \bot \ \text{ (and } \boldsymbol{I}^{\mathcal{C}}(k) \text{ is arbitrary)} \\ & \text{or} \quad \boldsymbol{I}^{\mathcal{C}}(k) \neq \bot, \boldsymbol{I}^{\mathcal{A}}(k) \neq \bot \text{ and } IO(\boldsymbol{I}^{\mathcal{A}}(k), \boldsymbol{I}^{\mathcal{C}}(k)) \text{ holds.}\end{aligned}$$

Refinement relative to IO is then defined as follows.

Definition 5. *Machine $\mathcal{C}$ refines $\mathcal{A}$ relative to IO, written $\mathcal{C} \sqsubseteq_{IO} \mathcal{A}$, if for every program $\underline{j}$ and every $\boldsymbol{I}^{\mathcal{C}} \in runs^{\mathcal{C}}(\underline{j})$ an abstract run $\boldsymbol{I}^{\mathcal{A}} \in runs^{\mathcal{A}}(\underline{j})$ exists, such that $\boldsymbol{I}^{\mathcal{C}} \sqsubseteq_{IO} \boldsymbol{I}^{\mathcal{A}}$ holds.*

The refinement definition allows to refine an abstract run, which calls a diverging operation (i.e., one where the precondition is violated) with a terminating run: the state $\boldsymbol{I}^{\mathcal{A}}(k)$ after the diverging operation (and all subsequent states) will be $\bot$, and match any concrete state.

Proofs of refinement are done with forward simulation:

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

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

Proof. For finite runs the proof is by induction over its length. Infinite runs require a simple diagonalization argument. □

5.2 Refinement of Submachines

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 for $\mathcal{L}$ in operations of $\mathcal{M}$ with calls to $\mathcal{K}$ gives a machine $\mathcal{C} := \mathcal{M}(\mathcal{K})$ that refines $\mathcal{A} := \mathcal{M}(\mathcal{L})$. The result needs one additional restriction compared to general refinement: relation LIO is the identity relation over the input and output parameters of $\mathcal{L}$ and $\mathcal{K}$. Otherwise calls could not just be replaced. The replacement of $\mathcal{L}$ by $\mathcal{K}$ in $\mathcal{M}(\mathcal{L})$ is defined as follows: The signature is $(SIG \setminus SIG^{\mathcal{L}}) \dot\cup SIG^{\mathcal{K}}$ and the initialization condition is

$$Init^{\mathcal{M}(\mathcal{K})}(ks, gs) \quad \leftrightarrow \quad Init^{\mathcal{K}} \wedge \exists\ ls.\ R(ls, ks) \wedge Init^{\mathcal{M}(\mathcal{L})}(ls, gs),$$

where ks is the local state of $\mathcal{K}$. The I/O correspondence IO extends LIO to the entire set of input/output parameters of $\mathcal{C}$ and $\mathcal{A}$ by identity.

In order to express the modularity of refinement on the level of intervals, we define the substitution $I' := I\{I\big|_{\mathcal{K}} \mapsto \boldsymbol{I}^{\mathcal{L}}\}$ of all calls to a submachine $\mathcal{K}$ by corresponding calls to $\mathcal{L}$ taken from $\boldsymbol{I}^{\mathcal{L}}$ assuming $\boldsymbol{I}^{\mathcal{L}}$ satisfies $I\big|_{\mathcal{K}} \sqsubseteq_{LIO} \boldsymbol{I}^{\mathcal{L}}$.

For each transition $\tau = (gs, ks, gs', ks')$ in I after k calls to the submachine, the corresponding transition of the substitution I' is $(gs, \boldsymbol{I}^{\mathcal{L}}(k), gs', ls')$ with

$$ls' = \begin{cases} \boldsymbol{I}^{\mathcal{L}}(k), & \text{if } \tau \text{ is not a call} \\ \boldsymbol{I}^{\mathcal{L}}(k+1), & \text{if } \tau \text{ is a call and } \boldsymbol{I}^{\mathcal{L}}(k+1) \neq \bot \\ \text{arbitrary}, & \text{if } \tau \text{ is a call and } \boldsymbol{I}^{\mathcal{L}}(k+1) = \bot \end{cases}$$

In the last case the $(k+1)$-th call to $\mathcal{L}$ did not terminate and we additionally demand that the interval I' is infinite and may be arbitrary after the call. According to $\sqsubseteq_{LIO}$, $\#\boldsymbol{I}^{\mathcal{L}} = \#I\big|_{\mathcal{K}}$ and $\boldsymbol{I}^{\mathcal{L}}$ reaches $\bot$ before $I\big|_{\mathcal{K}}$ (if at all). After lifting this substitution to an execution $\boldsymbol{I}$ of a program $\underline{j}$ it follows:

Lemma 2. *Given an execution $\boldsymbol{I}$ of a program $\underline{j}$ on $\mathcal{M}(\mathcal{K})$ and $\boldsymbol{I}^{\mathcal{L}}$ with $\boldsymbol{I}\big|_{\mathcal{K}} \sqsubseteq_{LIO} \boldsymbol{I}^{\mathcal{L}}$, then $\boldsymbol{I}' := \boldsymbol{I}\{\boldsymbol{I}\big|_{\mathcal{K}} \mapsto \boldsymbol{I}^{\mathcal{L}}\}$ is an execution of $\underline{j}$ on $\mathcal{M}(\mathcal{L})$ with $\boldsymbol{I} \sqsubseteq_{IO} \boldsymbol{I}'$. $\boldsymbol{I}'$ is a run if $\boldsymbol{I}$ is.*

Proof. The proof is by inspecting the (non-atomic) runs of each operation. For a single operation induction over rule complexity gives the desired result. □

Given these prerequisites we prove the compositionality theorem:

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

Proof. Let $\boldsymbol{I} \in runs^{\mathcal{C}}(\underline{j})$ be arbitrary. According to Lemma 1 $\boldsymbol{I}\big|_{\mathcal{K}}$ is a run of the submachine $\mathcal{K}$. By assumption there is a run $\boldsymbol{I}^{\mathcal{L}}$ of $\mathcal{L}$ with $\boldsymbol{I}\big|_{\mathcal{K}} \sqsubseteq_{LIO} \boldsymbol{I}^{\mathcal{L}}$. The matching abstract run then is $\boldsymbol{I}\{\boldsymbol{I}\big|_{\mathcal{K}} \mapsto \boldsymbol{I}^{\mathcal{L}}\}$ by Lemma 2. □

6 Related Work

In general our approach is based on (iterated) refinement, following the general idea of ASM refinement [4]. We prefer this over an approach that just annotates code with pre- and postconditions. With such an approach all the abstract layers would become ghost code and extra ghost state, cluttering the implementation with annotations. This would be particularly problematic for the Flash file system, since its refinement hierarchy is *deep*, at least a dozen layers and various submachines are necessary to conceptually isolate the relevant building blocks. Verifying that the whole implementation is a refinement of the POSIX specification in one step is practically infeasible.

The specific instance of refinement defined here is based on data refinement [13], in particular the contract-based approach of Z [28] (see [8] for other approaches, and [23] for a comparison to ASM refinement). 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 we think that they are easier to understand).

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 $\{\bot\} \times S_\bot$, but just $\{\bot\} \times \{\bot\}$ to preserve the meaning of $\bot$ as "nontermination" (not "unspecified"). [23] argues that for both embeddings the same refinements are correct. As a result the proof obligations for forward simulations are similar to those of Z refinement. As a minor difference our theory allows an operation to have diverging runs, even when its precondition is satisfied, though we have not exploited this in the Flash project (we always prove termination). The generalization results in the extra precondition $\langle\!|\alpha_j^A|\!\rangle$ `true` in the correctness proof obligation.

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. In [7] an analogous result is proved on a semantic level using relations of μ-calculus as context.

It should however be noted that the contract approach [28] itself is not sufficient for such a result, since it considers finite sequences of operation calls only, while our context (the main rule of an ASM) may be a loop calling operations of the submachine an infinite number of times. Considering finite runs only has the advantage that forward and backward simulation together give a complete proof technique. Here, as in most refinement definitions that consider infinite runs, backward simulation is not sound: it may result in implementing a terminating run of a rule of $\mathcal{M}(\mathcal{L})$ with a non-terminating run of $\mathcal{M}(\mathcal{K})$, when the abstract machine has infinite nondeterminism (i.e., has a **choose** from some infinite domain). Most of our ASMs have infinite nondeterminism.

The refinement concept discussed here differs from our earlier formalisation [22,24], and from Event-B [3] in that it uses *preconditions*, not *guards* (the earlier B formalism [2] had both preconditions and guards). Whether one needs one or the other concept is application dependent: when rules are "called" by the environment (as here), the precondition approach is appropriate, while applications, where the machine itself chooses a rule (e.g., an interpreter for a programming language, where the next rule is chosen according to the next statement to interpret), then the guard interpretation is appropriate.

The definition given here is on the one hand more liberal than the one in [24], as it allows one to implement a diverging operation on the abstract level with any run on the concrete level (since for $\boldsymbol{I}^A(k) = \bot$ any concrete state is allowed). On the other hand it is more strict, as it forbids general $m : n$ diagrams where m abstract operations are implemented with n concrete ones. The case $m > 1$ is disallowed here, since any sequence of submachine calls must be verified. The case $m = 0$ can be simulated by adding an abstract skip operation that does nothing. Diagrams with $n \neq 1$ are still implicitly possible, by using a concrete rule that takes n atomic steps (in the fine-grained semantics) to complete.

With respect to ASMs, our syntax only uses a fragment of the syntax available in [5]. 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 have two

drawbacks. Code generation would become more difficult, and simple symbolic execution rules would be precluded since parallel rules may have clashes. These require a complex axiomatization of update and consistency predicates, even when nondeterministic choice (that we often use for specification purposes) is omitted (see [26] and Chapter 8 of [5]).

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 [5], Chapter 4. $(s,\bot) \in [\![\mathtt{Op}]\!]$ corresponds to either a diverging computation of updates, or to the computation of an inconsistent set. Our largest fixpoint for the non-atomic semantics of **while** reduces to the least fixpoint definition 4.1.2 of **iterate** that is used to define the semantics of **while** with deterministic body in [5]. In general, using a largest fixpoint is unavoidable to characterize guaranteed termination for rules with infinite nondeterminism.

The non-atomic semantics we give in Def. 2 is based on Interval Temporal Logic (ITL [16,17]). We prefer this alternative over a structural operational semantics (SOS, [19]), since SOS must model an explicit stack of local variables which is unnecessary for a direct interval semantics.

The non-atomic semantics in this paper is a simplified version of the one we give in [**?**], which additionally handles interleaved concurrency and temporal operators.

Our definition of submachines is different from the one in [5]. A submachine there is a subrule that may be called within a rule, with the purpose to support mutual recursion in Turbo ASMs. These are similar to the calls of submachine operations, however, submachines as defined here are full ASMs (with initialization, signature etc.). Additionally, information hiding constraints have to be satisfied for modular refinement. To be able to check these constraints syntactically we use input and output parameters passed by value, whereas subrules in [5] use call by name. This extension does not give additional expressivity: a declaration $\mathtt{Op}(\underline{x},\underline{y})\{\alpha\}$ could be replaced $\mathtt{Op}(;\underline{x},\underline{y})\{\alpha\}$ using reference parameters only. Calls $\mathtt{Op}(\underline{t};\underline{z})$ for a submachine operation with declaration would have to be replaced with **let** $\underline{in} = \underline{t}$ **in** $\mathtt{Op}(;\underline{in},\underline{z})$.

The ASM formalism is also strong enough such that preconditions are definable. A rule *RULE* working on dynamic functions $f_1,\ldots f_n$ with precondition *pre* is equivalent to the extended rule **if** *pre* **then** *RULE* **else** *CHAOS* where

CHAOS = **choose** *diverge*? **in if** *diverge*? **then abort**
else *RANDOM*(f_1)
...
RANDOM(f_n)

and

RANDOM(f_i) = **forall** $args_i$ **choose** *val* **in** $f_i(args_i) \coloneqq val$

Rule *CHAOS* either diverges (when *diverge*? is true) or chooses a random next state by overwriting each f_i with a new function in *RANDOM*(f_i).

Event-B has two decomposition concepts for machines that roughly correspond to interleaved [1] and synchronous parallel execution of rules [6]. 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 (although a construction with program counters and explicit call/return events for subrules may be possible).

7 Conclusion

We have defined a refinement theory for ASMs with submachines, which respect information hiding. The theory has been key to enable modular, incremental development of the Flash case study.

So far we have used forward simulations for our proof only. As noted in related work, backward simulation is not a sound proof technique in the presence of infinite runs. A completeness proof will therefore be possible only along the lines of [24], by replacing **choose** with choice functions, but such a proof is still future work.

Although we have no need for guards in the Flash case study (the toplevel POSIX specification has total operations, all intermediate layers have preconditions), it would be interesting to analyze, whether our refinement definition for submachines is compatible with the main machine having guards.

Finally, it should be noted that our definition of refinement does not solve all problems in the Flash case study. One important extension is necessary to deal with power failures and recovery. A paper on this issue based on the same semantic setting (with the idea that runs of a rule may be aborted in any intermediate state) is currently in preparation. Another important issue, which we will have to consider, is that the actual implementation uses concurrency to do work in the background: As an example, actually erasing blocks is done in a concurrent thread that calls back to the main thread, when it has finished.

References

1. J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77, 2007.
2. Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.
3. Jean-Raymond Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.
4. E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15(1–2):237–257, 2003.
5. E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis*. Springer, 2003.
6. Michael Butler. Decomposition Structures for Event-B. In *IFM '09: Proceedings of the 7th International Conference on Integrated Formal Methods*, pages 20–38, Berlin, Heidelberg, 2009. Springer-Verlag.
7. W. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

8. J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. FACIT. Springer, 2001. second, revised edition 2014.
9. G. Ernst, J. Pfähler, and G. Schellhorn. Web presentation of the Flash Filesystem. `https://swt.informatik.uni-augsburg.de/swt/projects/flash.html`, 2014.
10. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. A Formal Model of a Virtual Filesystem Switch. In *Proc. of Software and Systems Modeling (SSV)*, pages 33–45, 2012.
11. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a Virtual Filesystem Switch. In *Proc. of Verified Software: Theories, Tools, Experiments*, volume 8164, pages 242–261. Springer, 2014.
12. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
13. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. of the European symposium on programming on ESOP 86*, pages 187–196. Springer-Verlag New York, Inc., 1986.
14. A. Hunter. A brief introduction to the design of UBIFS. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`, 2008.
15. R. Joshi and G.J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2), June 2007.
16. B. Moszkowski. *Executing Temporal Logic Programs*. Cambr. Univ. Press, 1986.
17. B. Moszkowski. An automata-theoretic completeness proof for Interval Temporal Logic. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 223–234. Springer-Verlag, 2000.
18. J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Formal Specification of an Erase Block Management Layer for Flash Memory. In *Hardware and Software: Verification and Testing*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013.
19. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.
20. G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference*, pages 4186–4199. IEEE Computer Society, 2005.
21. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II, pages 13–39. Kluwer, Dordrecht, 1998.
22. G. Schellhorn. Verification of ASM Refinements Using Generalized Forward Simulation. *Journal of Universal Computer Science (J.UCS)*, 7(11):952–979, 2001. URL: `http://www.jucs.org`.
23. G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, 336(2–3):403–435, 2005.
24. G. Schellhorn. Completeness of Fair ASM Refinement. *Science of Computer Programming, Elsevier*, 76, issue 9:756 – 773, 2009.
25. G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a Verified Flash File System. In *Proc. of ABZ 2014*, LNCS. Springer, 2014.
26. R. F. Stärk and S. Nanchen. A Complete Logic for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 7 (11):981 – 1006, 2001.
27. 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).
28. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.