SRI International

CSL Technical Report SRI-CSL-02-04 • 22nd October 2004

A Semantic Embedding of the $\mathbf{A_g}$ Dynamic Logic in *PVS*

Carlos López Pombo[1]
Sam Owre[2]
Natarajan Shankar[3]

The work of the first author was supported by the School of Sciences, University of Buenos Aires through FOMEC project 359, and SRI International. The other authors were supported by NASA contract number NAS1-0079 and NSF contract number CCR-0082560.

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Acknowledgements

Carlos Lopez Pombo:

First of all I want to thank Natarajan Shankar for making my stay at the Formal Methods group of the SRI Computer Science Laboratory possible. Sam Owre, Natarajan Shankar and Marcelo Frías were the people responsible for making my time at SRI fruitful by giving me guidance and support.

Thanks to Josh Levy for providing me the vital element, *yerba mate*, without it I would never found a way to survive four months. And thanks to Sam for sharing with me those *mates*. For those who believe in traditions it is very important to find somebody willing to share them.

Finally I want to thank all CSL staff, but specially to John Rushby and Hassen Saïdi for their hospitality and support during my stay.

The list of the people from Argentina that never let me down and worked to make my travel possible is so large that I could never fit it in this page, thanks to all of you.

Abstract

$\mathbf{A_g}$ is a specification language presented as a syntactic sugaring of the First-Order Dynamic Logic of Fork Algebras. This language is particularly attractive due to its expressive power, easy-to-understand semantics, and the existence of a complete deductive system. We will briefly present the language together with a complete deductive calculus and some theoretical results about its expressive power. We then show how the higher-order logic theorem prover *PVS* can be used to build a semantic framework for reasoning about specifications written in $\mathbf{A_g}$ by encoding the semantics within this powerful general-purpose tool.

We then illustrate how this semantic embedding can be used by means of a case study of a cache system specification.

Contents

List of Figures

Chapter 1

Introduction

Due to the increasing complexity of software systems, the application of formal methods to prove the correctness of specifications and designs is becoming more useful in avoiding system failures. One of the main disadvantages of formal methods is that the application of the available tools requires significant knowledge and skill in order to get effective communication between the human and the computer.

There is much ongoing work attempting to make formal methods applicable by people that do not have a strong formal methods education. An example of this is *Alloy* [Jac02, JSS01], a widely used language that allows the level of automation to be tailored in the process of development and validation of specifications.

This report addresses the problem of mechanizing the verification of specifications written in $\mathbf{A_g}$ [FPB02, FBP01]. This specification language is a relational framework that is intended to be the kernel of a tool suitable for specification and verification of industrial applications.

The language $\mathbf{A_g}$ is the First-Order Dynamic Logic of Fork Algebra terms. It has a quite simple syntax based on First-Order Dynamic Logic, and an easy-to-understand semantics at the term level based on binary relations. Results on the interpretability of several logics into the equational calculus of Fork Algebras illustrate the expressive power of this language. This expressiveness is extremely useful for the specification of different views of a system, each with a different logic as needed. We believe that these features make it a good choice in the specification and design stages of a software project development.

We use the classical Higher-Order Logic of *PVS* as a means of embedding $\mathbf{A_g}$ semantics with the aim of using its capabilities to construct specifications and develop readable proofs of the properties involved. In Chapter 3 we present the highlights of this powerful theorem prover that has been successfully applied to industrial problems in many areas, including microcode verification [MS95], fault-tolerance algorithms [Min00], and Java code verification [vdBJ01].

Fork Algebras were introduced by Haeberer and Veloso [HV91] when looking for a framework for system specification, construction, and verification. The calculus allows binary relations to be built from the predetermined constants "0", "1" and "$1'$", plus operations such as sum "$+$", product "$\cdot$", complement "$^-$", converse "$\breve{\ }$", composition "$;$", and *fork* "∇", (to be defined later). The calculus of fork algebras has been used as an intermediate framework for interpretation of different logics. Previous work due to Frías, Maibaum, Orlowska, *et al.,* showed that several logics such as classical first-order logic [VHF95, FBH97], a wide class of modal logics [FO98], and propositional [FO98] and first-order dynamic logic [FBM01], could be interpreted within fork algebras.

Dynamic Logic is the name given to a family of logic systems originally meant to formalize concepts such as correctness of specifications and programs[Pra76]. As mentioned by Harel *et al.,* in [HKT00], other activities fall into the category of problems addressed by this framework, such as synthesizing programs from specifications or determining the equivalence of programs.

First-Order Dynamic Logic is a framework composed of three ingredients, first-order predicate logic, modal logic, and the algebra of regular events. The result is a system with practical and theoretical interest because of the problems it can address.

In Dynamic Logic, programs are syntactic objects that are meant to change the values of variables and consequently the truth value of the formulas. For example, the program "$x := x+1$", changes the value of the formula "x is even". In traditional Dynamic Logic, programs are constructed from assignments "$x := t$", composition ";", choice "+" and iteration "*". Formulas are constructed, as in traditional first-order predicate logic, from predicate symbols applied to terms, negation "$\neg$", disjunction "$\vee$" and existential quantification "$\exists$". The relation between programs and formulas is established by using the mixed operators test "?", that takes a formula and yields a program and the possibility operator "$\langle\rangle$", that takes a program and a formula and yields a formula.

This report is organized as follows. In Chapter 2, we present the syntax, semantics, and complete deductive calculus for $\mathbf{A_g}$, together with a description of the expressive power of the language. In Chapter 3, we present the *PVS* system, particularly those features that are helpful in the implementation of a proof checker for this language.

In Chapter 4, we show how *PVS* [OSRSC01c, OSRSC01b, OSRSC01a] can be used as a semantic framework for reasoning about specifications written in $\mathbf{A_g}$ by encoding its semantics within this powerful general-purpose tool.

In Chapter 5, we develop an example of a cache system [FBP01], giving the complete specification of this system and its desired behavior illustrating how this framework can be used to build and verify such specifications.

We finish this work by presenting our conclusions in Chapter 6.

There are two appendices. Appendix A discusses some esthetical characteristics of presenting $\mathbf{A_g}$ proofs within the PVS proof checker. Appendix B shows a graph that models the import chain of files that constitute the framework in which an $\mathbf{A_g}$ specification can be verified using PVS.

Chapter 2

The $\mathbf{A_g}$ Specification Language

As mentioned earlier, $\mathbf{A_g}$ has its origins in the equational calculus of fork algebras, which is presented in the first section below. We will complete the definition of the language in Section 2.2 by presenting the first-order dynamic logic of fork algebras, and show some interesting results.

2.1 Syntactic and Semantic Foundations of $\mathbf{A_g}$: Fork Algebras

The calculus of fork algebras (also called *fork calculus*), is an equational calculus. In order to give a full description of the calculus, it suffices to provide the language, axioms and inference rules. Formulas are identities between terms. Terms are built from variables, the constants "0", "1" and "$1'$", the unary operations "$^-$" (*complement*) and "$\smile$" (*converse*), and the binary operations "$+$" (*sum*), "$\cdot$" (*product*), "$;$" (*composition*) and "∇" (*fork*). The inference rules are those of equational logic [BS81]. The axioms are those of the Boolean calculus, $\langle 0, 1, ^-, +, \cdot \rangle$, plus some identities that specify the behavior of "$\smile$", "$;$" and "∇".

Definition 2.1 *Syntax of the fork calculus.*

- 0, 1 *and* $1'$ *are* Terms,
- *If* t *is a* Term, *then* $\overline{t}$ *and* $t^{\smile}$ *are* Terms,
- *If* t *and* t' *are* Terms, *then* $t + t'$, $t \cdot t'$, $t \,;t'$ *and* $t \nabla t'$ *are* Terms, *and*
- *If* t *and* t' *are* Terms, *then* $t = t'$ *is a* Formula.

Definition 2.2 *Deductive system for the fork calculus. The deductive system for the fork calculus consists of the inference rules of the equational calculus. the Boolean algebra axioms for* $\langle 0, 1, ^-, +, \cdot \rangle$, *and the following identities:*

- $x\,;(y\,;z) = (x\,;y)\,;z$ *(associativity of "$;$"),*

- $(x + y)\,;z = x\,;z + y\,;z$ *(distributivity of "$\,;$" over "$+$")*,
- $(x + y)^{\smile} = x^{\smile} + y^{\smile}$ *(distributivity of "$^{\smile}$" over "$+$")*,
- $(x^{\smile})^{\smile} = x$ *(involution of "$^{\smile}$")*,
- $x\,;1' = x$ *("$1'$" is neutral for "$\,;$")*,
- $(x\,;y)^{\smile} = y^{\smile}\,;x^{\smile}$ *(distributivity of "$^{\smile}$" over "$\,;$")*,
- $(x\,;y) \cdot z = 0$ *iff* $(z\,;y^{\smile}) \cdot x = 0$ *iff* $(x^{\smile}\,;z) \cdot y = 0$,
- $x\nabla y = (x\,;(1'\nabla 1)) \cdot (y\,;(1\nabla 1'))$
- $(w\nabla x)\,;((y\nabla z)^{\smile}) = (w\,;y^{\smile}) \cdot (x\,;z^{\smile})$
- $(((1'\nabla 1)^{\smile})\nabla((1\nabla 1')^{\smile})) + 1' = 1'$

Then we add, to this set of axioms, the following two formulas. The first one requires the models to be point-dense. Quoting Maddux in [Mad91]:

> "A relation algebra is said to be point-dense if every nonzero element below the identity contains a point."

A point is a relation r that satisfies $\neg(r = 0) \wedge x\,;1\,;x \leq 1'$. So the new axiom ends up being

$$(\forall x)(\neg(x = 0) \wedge x \leq 1' \Rightarrow (\exists y)(\neg(y = 0) \wedge y\,;1\,;y \leq 1' \wedge y \leq x)) \tag{2.1}$$

The second axiom is known as the Tarski rule and requires the models to be simple.

$$(\forall x)(\neg(x = 0) \Rightarrow 1\,;x\,;1 = 1) \tag{2.2}$$

This fork calculus gives rise to the class of point-dense simple abstract fork algebras, presented next.

Definition 2.3 *A point-dense simple abstract fork algebra is an algebraic structure* $\langle R, +, \cdot, ^{-}, 0, 1, \,;, 1', \,^{\smile}, \nabla\rangle$ *where R is a set, "$+$", "$\cdot$", "$\,;$", "∇" are binary operations over R; "$^{-}$", "$^{\smile}$" are unary operations over R; "0", "1", "$1'$" are distinguished elements of R, and this is a model of the identities presented in Definition 2.2, and the Formulas 2.1 and 2.2.*

Proper fork algebras are those in which the universe is made of binary relations on a set A closed under a binary function $\star$. The class of proper fork algebras to be used in this report requires only that $\star$ is injective and was introduced by Frias *et al.* in [FBHV95].

In a proper fork algebra "0" is the empty relation, "1" is the universal relation, "$+$", "$\cdot$", and "$\,;$" are the sum, product, and composition of binary relations, "$^{\smile}$" is transposition, "$^{-}$" is the complement relation, "$1'$" is the identity relation, and "∇" is defined as follows:

$$R\nabla S = \{\, \langle x, \star(y, z)\rangle : x \; R \; y \wedge x \; S \; z \,\} \;\; .$$

Figure 2.1 shows the definition of "∇" (*fork*) applied to relations R and S.

Now we will define proper fork algebras formally. To do this, we introduce the class of star proper fork algebras.

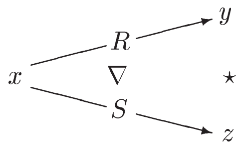

Figure 2.1: Definition of "∇" (*fork*) applied to relations R and S.

Definition 2.4 *A* star proper fork algebra *is a two sorted structure* $\langle R, U, +, 0, \cdot, E, ^-, ;, ^\smile, 1', \nabla, \star\rangle$, *where "0", "E" and "1'" are constants, "$^-$" and "$^\smile$" are unary operators, and "+", "·", ";" and "∇" are binary operators satisfying the following conditions:*

- $\langle R, +, 0, \cdot, E, ^-, ;, ^\smile, 1'\rangle$ *is an* algebra of binary relations *on* U*,*
- $\star : U^2 \to U$ *is an injective function when its domain is restricted to* E*,*
- R *is closed by the operation* ∇*, defined as follows:*

$$S \nabla T = \{\langle x, y \star x\rangle | x\ S\ y \wedge x\ T\ z\}.$$

The class *star proper fork algebras* is denoted as $\star$PFA.

Definition 2.5 *The class of* proper fork algebras *is the class obtained by taking reducts to the similarity type* $\langle R, +, 0, \cdot, E, ^-, ;, ^\smile, 1', \nabla\rangle$ *of the algebras in the class* $\star$*PFA.*

The class containing those proper fork algebras that satisfy axioms 2.1 and 2.2 will be called *point-dense simple proper fork algebras.*

Notice that the main reason for the use of *fork*, besides its practical use in the definition of *disjoint union*, *first projection*, and *second projection* (to be shown), can be found at the end of Tarski's paper [Tar41], where he not only developed the *elementary theory of binary relations*, he also presented the *relation algebras* as the restriction of this theory to those formulas where no individual variables appear. He also posed some questions involving the representability of relation algebras. These were later answered negatively by Roger Lyndon in [Lyn50, Lyn56] by exhibiting a non-representable relation algebra (i.e., a relation algebra that is not isomorphic to any algebra of binary relations). The immediate consequence of this result is that there exist properties valid in all algebras of binary relations which can be false in some relation algebras. On the other hand, as we said in a previous paragraph, the extension of Tarski's relation algebras with fork, or fork algebras, were proved representable, [FBHV95]. Actually, point-density also guarantees representability by itself; nevertheless fork is better suited because it can be axiomatized equationally.

It is clear that every point-dense simple proper fork algebra satisfies the axioms, and therefore is a point-dense simple abstract fork algebra. The other inclusion follows from the next representability theorem that can be proved by mimicking the proof presented in [FBHV95] for the representability of fork algebras, but considering the fact that point-dense relation algebras are representable, proved in [Mad91].

Theorem 2.1 *Given a point-dense simple abstract fork algebra* $\mathfrak{A}$ *there exists a point-dense simple proper fork algebra* $\mathfrak{B}$ *isomorphic to* $\mathfrak{A}$*.*

Since this theorem implies that the axiomatization provided for the class of point-dense simple abstract fork algebras is complete with respect to the formulas valid in the point-dense simple proper fork algebras, it provides the other inclusion and is the key argument for proving the completeness of $\mathbf{A_g}$.

An immediate consequence of this theorem is that the equational (and also first-order) theory of point-dense simple abstract fork algebras and the point-dense proper fork algebras are the same (of course, with respect to the language of the abstract fork algebras).

From now on we will refer to the point-dense simple abstract (proper) fork algebras simply as abstract (proper) fork algebras, but keeping in mind the axioms we added to assure point-density and simplicity of the models.

We now recall some notation that will be used often in this work, specially in Chapter 5 where we present a case study.

We denote by "$\leq$" the ordering relation induced by the Boolean algebra underlying the fork algebra. Given a relation R, $dom(R)$ denotes the term $(R\,;R^{\smile})\cdot 1'$. Notice that for any R, $dom(R)$ is a partial identity, i.e., a relation contained in $1'$. Since partial identities define domains, the *complement domain* of a partial identity I is denoted by $\neg I$ and defined as $\overline{I}\cdot 1'$. A relation R is called *functional* if it satisfies the condition $R^{\smile}\,;R \leq 1'$, and *one-to-one* if it satisfies $R\,;R^{\smile} \leq 1'$. Notice that when viewed in a proper fork algebra, $dom(R)$ is indeed a subset of the identity binary relation. Similarly, functional relations are indeed functions and one-to-one relations are indeed injective. We also define projection π as the relation $(1'\nabla 1)^{\smile}$ and projection ρ as $(1\nabla 1')^{\smile}$ (notice that π acts as the first projection and ρ as the second one).

Given binary relations R and S, the disjoint union of R and S, denoted by $R \oplus S$, is defined as follows:[1]

$$R \oplus S = \begin{array}{c} \pi\,;C_0 \\ \nabla \\ \rho\,;(R\,;inl) \end{array} \;;\; \rho \quad + \quad \begin{array}{c} \pi\,;C_1 \\ \nabla \\ \rho\,;(S\,;inr) \end{array} \;;\; \rho$$

In this context, C_0 and C_1 are constant relations with only one pair, for instance, the pair $\langle c_0, c_0\rangle$ in the case of C_0 and $\langle c_1, c_1\rangle$ in the case of C_1, with c_0 and c_1 distinguished elements. The relations inl and inr are the left and right injections of the disjoint union.

The disjoint union can be understood in the following way. First, it tags input and output elements in R in *red*, i.e., if $\langle x, y\rangle \in R$, then the pair becomes $\langle \star(red, x), \star(red, y)\rangle$. Similarly, *blue* is used for those elements in S. Once domain and range elements are tagged, $R \oplus S$ performs the union of the tagged relations R and S. In this case, c_0 is *red*, c_1 is *blue*, and the relation inl, given an input element, returns the element tagged in *red* and inr, the element tagged in *blue*. Notice that the converses of inl and inr remove tags from appropriately tagged objects.

Because of the importance of the fork algebras in computer sciences as a research field itself, we refer the interested reader to [Frí02].

[1] Because of its complexity, this definition is presented in a two dimensional *Begriffsschrift*-like diagram.

2.2 $\mathbf{A_g}$: First-Order Dynamic Logic of Fork Algebras

Classical logic is a language suitable for describing properties in a static universe. In fact, there is a single universe that fixes the meaning of the propositional atoms and consequently of functions and predicates. Classical logics are not well suited for expressing properties that can change their truth value in a dynamic way, for example, invariants of programs.

Modal logics introduce the notion of possible worlds, and truth values are fixed but only within the scope of a world. A consequence of this is that in modal logics the propositional atoms, functions, and predicates can take different values in different worlds.

The traditional semantics of modal logics are Kripke models [Ham88]. These structures consist of a set of worlds, and give meaning to propositional atoms, functions, and predicates relative to a given world, and one or more accessibility relations between worlds. Every modal logic introduces one or more *modal operators*. An example of this is given by the formula "$\Box f$"; it is valid in a world "w" if and only if in every world, accessible from "w", the formula "f" is valid.

We are interested in a particular modal logic called first-order dynamic logic (FODL). This modal logic allows the expression of program behavior taking as atomic programs assignments of the form "$x := t$", where x is a variable symbol and t is a term, and the "SKIP" program. More complex programs can be constructed by the operations "?", that take as argument a FODL formula and test for its truth value, "+" and ";", that take two programs and perform a non-deterministic choice and a sequential composition, respectively, and "*" that takes a program and performs an unbounded, but finite, number of iterations of it.

In fact, we will use an extension of FODL that makes use of atomic actions and metavariables. To make the distinction with traditional FODL, this extension will be called FODLwA.

Definition 2.6 *Syntax of* FODLwA.

Given sets of symbols C, M, V, A, P, and F for constants, metavariables, variables, atomic actions, predicates, and functions (with their arity), we define Terms, Formulas, *and* Programs *over the signature $\Sigma = \langle C, M, V, A, P, F \rangle$ as follows.*

Terms

- If $c \in C$ then c is a *Term*,
- If $m \in M$ then m is a *Term*,
- If $v \in V$ then v is a *Term*,
- If $t \in F$ and $t_1, \ldots, t_n$ are *Terms* then $t(t_1, \ldots, t_n)$ is a *Term*, where n is the arity of t,

Formulas

- TRUE is a *Formula*,
- If $p \in P$ and $t_1, \ldots, t_j$ are *Terms* then $p(t_1, \ldots, t_j)$ is a *Formula*, where j is the arity of p,

- If f is a *Formula* then $\neg f$ is a *Formula*,
- If f and g are *Formulas* then $f \Rightarrow g$ is a *Formula*,
- If $v \in V$ and f is a *Formula* then $(\exists v)f$ is a *Formula*,
- If f is a *Formula* and P is a *Program* then $\langle P \rangle f$ is a *Formula*,

Programs

- If f is a *Formula* with no reference to metavariables then $f?$ is a *Program*,
- SKIP is a *Program*,
- If $a \in A$ and f and g are *Formulas* then $a(f, g)$ is a *Program*,
- If $v \in V$ and t is a *Term* with no reference to metavariables then $v := t$ is a *Program*,
- If Q and Q' are *Programs* then
  - $Q + Q'$ is a *Program*,
  - $Q; Q'$ is a *Program*, and
- If Q is a *Program* then Q^* is a *Program*.

Now, a formula will be well-formed if and only if no metavariable symbol appears in any program involved. It is easy to construct a recursive function that checks for this property.

Note that all other formulas and programs can be obtained from the existing ones. It is important to recall that the dual definition of the "eventually" modal statement, $\langle P \rangle f$, gives rise to the "necessarily" modal statement, defined as $[P]f = \neg\langle P \rangle \neg f$. Note also that atomic actions are built from two formulas, say for instance $a(f, g)$, where f is the pre-condition and g the post-condition of the atomic action a. Carroll Morgan's specification statement [Mor88] also specifies actions in terms of pre-conditions and post-conditions, but additionally includes a frame condition stating which variables are allowed to be modified.

Definition 2.7 *Consider the* FODLwA *signature* $\Sigma = \langle C, M, V, A, P, F \rangle$. *Let* $\mathcal{C}$ *be a set that acts as the carrier of the logic, and define a* world w *to be a function from* V *to* $\mathcal{C}$. *The structure* $\mathfrak{A}$ *is* $\langle A_{\mathfrak{A}}, m_{\mathfrak{A}} \rangle$, *where* $A_{\mathfrak{A}}$ *is the set of all* worlds, η *is a function from* M *to* $\mathcal{C}$, *and* $m_{\mathfrak{A}}$ *is the meaning function for terms, formulas, and programs, defined as follows.*

Meaning Function for Terms

- If $c \in C$, then $m_{\mathfrak{A}}(c) \in \mathcal{C}$,
- If $m \in M$, then ${m_{\mathfrak{A}}}^{w,\eta}(m) = \eta(m)$,
- If $v \in V$, then ${m_{\mathfrak{A}}}^{w,\eta}(v) = w(v)$,
- If $f \in F$ and $t_1, \ldots, t_n$ are *Terms*, where n is the arity of f, then
  ${m_{\mathfrak{A}}}^{w,\eta}(f(t_1, \ldots, t_n)) = m_{\mathfrak{A}}(f)({m_{\mathfrak{A}}}^{w,\eta}(t_1), \ldots, {m_{\mathfrak{A}}}^{w,\eta}(t_n))$
  where $m_{\mathfrak{A}}(f)$ is a function in $\mathcal{C}^n \to \mathcal{C}$

Meaning Function for Formulas

- $m_{\mathfrak{A}}{}^{\eta}(\text{TRUE}) = A_{\mathfrak{A}}$,
- If $p \in P$ and $t_1, \ldots, t_n$ are *Terms*, where n is the arity of p, then
  $m_{\mathfrak{A}}{}^{\eta}(p(t_1, \ldots, t_n)) = \{w \in A_{\mathfrak{A}} \mid m_{\mathfrak{A}}(p)(m_{\mathfrak{A}}{}^{w,\eta}(t_1), \ldots m_{\mathfrak{A}}{}^{w,\eta}(t_n))\}$,
  where $m_{\mathfrak{A}}(p)$ is a n-ary relation on $\mathcal{C}$,
- If f is a *Formula*, then
  $m_{\mathfrak{A}}{}^{\eta}(\neg f) = \{w \in A_{\mathfrak{A}} \mid w \notin m_{\mathfrak{A}}{}^{\eta}(f)\}$,
- If f and g are *Formulas*, then
  $m_{\mathfrak{A}}{}^{\eta}(f \Rightarrow g) = \{w \in A_{\mathfrak{A}} \mid w \in m_{\mathfrak{A}}{}^{\eta}(f) \text{ implies } w \in m_{\mathfrak{A}}{}^{\eta}(g)\}$,
- If $v \in V$ and f is a *Formula*, then
  $m_{\mathfrak{A}}{}^{\eta}((\exists v)f) = \{w \in A_{\mathfrak{A}} \mid \exists a \in \mathcal{C} : w|_a^v \in m_{\mathfrak{A}}{}^{\eta}(f)\}$,
- If f is a *Formula* and P is a *Program*, then
  $m_{\mathfrak{A}}{}^{\eta}(\langle P\rangle f) = \{w \in A_{\mathfrak{A}} \mid \exists w' \in A_{\mathfrak{A}} : (w, w') \in m_{\mathfrak{A}}{}^{\eta}(P) \text{ and } w' \in m_{\mathfrak{A}}{}^{\eta}(f)\}$

Meaning Function for Programs

- If f is a *Formula* with no reference to metavariables, then
  $m_{\mathfrak{A}}{}^{\eta}(f?) = \{(w, w') \in A_{\mathfrak{A}} \times A_{\mathfrak{A}} \mid w = w' \text{ and } w \in m_{\mathfrak{A}}{}^{\eta}(f)\}$,
- $m_{\mathfrak{A}}{}^{\eta}(\text{SKIP}) = \{(w, w') \in A_{\mathfrak{A}} \times A_{\mathfrak{A}} \mid w = w'\}$,
- If f and g are *Formulas*, then
  $m_{\mathfrak{A}}{}^{\eta}(a(f, g)) = \{(w, w') \in A_{\mathfrak{A}} \times A_{\mathfrak{A}} \mid \text{for all } \eta' : [M \to \mathcal{C}]$
  $w \in m_{\mathfrak{A}}{}^{\eta'}(f) \text{ and } w' \in m_{\mathfrak{A}}{}^{\eta'}(g)\}$,
- If $v \in V$ and t is a *Term* with no reference to metavariables, then
  $m_{\mathfrak{A}}{}^{\eta}(v := t) = \{(w, w') \in A_{\mathfrak{A}} \times A_{\mathfrak{A}} \mid w' = w|_t^v\}$,
- If P and P' are *Programs*, then
  - $m_{\mathfrak{A}}{}^{\eta}(P + P') = \{(w, w') \in A_{\mathfrak{A}} \times A_{\mathfrak{A}} \mid$
    $(w, w') \in m_{\mathfrak{A}}{}^{\eta}(P) \text{ or } (w, w') \in m_{\mathfrak{A}}{}^{\eta}(P')\}$,
  - $m_{\mathfrak{A}}{}^{\eta}(P; P') = \{(w, w') \in A_{\mathfrak{A}} \times A_{\mathfrak{A}} \mid \exists w'' \in A_{\mathfrak{A}} :$
    $(w, w'') \in m_{\mathfrak{A}}{}^{\eta}(P) \text{ and } (w'', w') \in m_{\mathfrak{A}}{}^{\eta}(P')\}$,
- If P is a *Program*, then
  $m_{\mathfrak{A}}{}^{\eta}(P^*) = \mu\mathcal{X} : \{(w, w') \in A_{\mathfrak{A}} \times A_{\mathfrak{A}} \mid w = w' \text{ or } \exists w'' \in A_{\mathfrak{A}} :$
  $(w, w'') \in m_{\mathfrak{A}}{}^{\eta}(P) \text{ and } (w'', w') \in \mathcal{X}\}$.

Note that in the case of constant, function and predicate symbols, the value of $m_{\mathfrak{A}}$ is defined in advance and does not depend on the *world* or the metavariable assignment, and in the case of functions and programs the value of $m_{\mathfrak{A}}$ does not depend on the *world*.

Definition 2.8 *A formula f is said to be valid for structure $\mathfrak{A}$ and a given state w, written $\mathfrak{A}, w \models f$, if and only if for all $\eta : [M \to \mathcal{C}]$, $w \in m_{\mathfrak{A}}{}^{\eta}(f)$. f is said to be valid for $\mathfrak{A}$ if and only if for all worlds w, $\mathfrak{A}, w \models f$, written $\mathfrak{A} \models f$. f is said simply valid if for all structures $\mathfrak{A}$ and worlds w, $\mathfrak{A}, w \models f$, written $\models f$.*

Consider now the language of traditional FODL with *wildcard assignment* [HKT00, chapter 11]. The *wildcard assignment* is a particular statement that performs the assignment of an unknown value to a variable, and is defined as $\langle x := *\rangle f = (\exists x) f$. We will show that FODL and FODLwA are equally expressive.

It is easy to see that every FODL formula can be translated to FODLwA, and as the semantics have been defined consistently we can state the following theorem:

Theorem 2.2 *Given a* FODL *formula f,*

$$\models_{\mathsf{FODL}} f \textit{ implies } \models_{\mathsf{FODLwA}} f$$

Note that the translation from FODLwA formulas to FODL is quite straightforward, except that occurrences of metavariables must be removed, and atomic actions must be expressed in terms of traditional programs.

Definition 2.9 *Given sets of symbols C, M, V, A, P, and F for constants, metavariables, variables, atomic actions, predicates and functions (with their arity); consider the signature $\Sigma = \langle C, M, V, A, P, F \rangle$:*

Formula Translations

- $tr(\mathrm{TRUE}) = \mathrm{TRUE}$,
- If $p \in P$ and $t_1, \ldots, t_n$, with n the arity of p, are *Terms*, then $tr(p(t_1, \ldots, t_n)) = p(t_1, \ldots, t_n)$,
- If f is a *Formula*, then $tr(\neg f) = \neg tr(f)$,
- If f and g are *Formulas*, then $tr(f \Rightarrow g) = tr(f) \Rightarrow tr(g)$,
- If $v \in V$ and f is a *Formulas*, then $tr((\exists v) f) = (\exists v) tr(f)$,
- If f is a *Formula* and P is a *Program*, then

  $$tr(\langle P \rangle f) = (\forall y_0 \ldots y_k)\langle tr(P)\rangle tr(f),$$

  where y_i are the variable symbols introduced to replace the metavariable symbols in the atomic actions that occur in P.

Program Translations

- If f is a *Formula* with no reference to metavariables, then $tr(f?) = tr(f)?$,
- $tr(\mathrm{SKIP}) = \mathrm{SKIP}$,

- If $a \in A$ and f and g are *Formulas*, then $tr(a(f,g)) = (f|^{m_i}_{y_{m_i}}?); x_j := *; (g|^{m_i}_{y_{m_i}}?)$, where y_{m_i} is a new variable symbol introduced to replace the metavariable m_i, and all x_j are the variables mentioned in the original formula.

- If $v \in V$ and t is a *Term* with no reference to metavariables, then $tr(v := t) = v := t$,

- If P and P' are *Programs*, then
  - $tr(P + P') = tr(P) + tr(P')$,
  - $tr(P; P') = tr(P); tr(P')$, and

- If P is a *Program*, then $tr(P^*) = tr(P)^*$.

Theorem 2.3 *Given a* FODLwA *formula* f,

$$\models_{\mathsf{FODLwA}} f \textit{ implies } \models_{\mathsf{FODL}} (\forall y_{m_i}) tr(f|^{m_i}_{y_{m_i}})$$

where y_{m_i} *is a new variable symbol introduced to replace the metavariable* m_i.

Theorem 2.2 is very simple. Theorem 2.3 follows from the fact that every FODLwA model for f is isomorphic to a FODL model for the translation of f. Contact the authors for complete proofs of these theorems.

From the last two theorems and [HKT00, theorem 14.7] it follows that *S2*, one of the various deductive systems for FODL, is a sound and complete deductive system for FODLwA. We now present *S2*.

Definition 2.10 *The deductive system* S2 *is the set of axioms for classical first-order logic, enriched by the following formulas:*

- $\langle P\rangle f_0 \wedge [P] f_1 \Rightarrow \langle P\rangle (f_0 \wedge f_1)$,
- $\langle P\rangle (f_0 \vee f_1) \Leftrightarrow \langle P\rangle f_0 \vee \langle P\rangle f_1$,
- $\langle P_0 + P_1\rangle f \Leftrightarrow \langle P_0\rangle f \vee \langle P_1\rangle f$,
- $\langle P_0; P_1\rangle f \Leftrightarrow \langle P_0\rangle\langle P_1\rangle f$,
- $\langle f_0?\rangle f_1 \Leftrightarrow f_0 \wedge f_1$,
- $f \vee \langle P\rangle\langle P^*\rangle f \Rightarrow \langle P^*\rangle f$,
- $\langle P^*\rangle f \Rightarrow f \vee \langle P^*\rangle(\neg f \wedge \langle P\rangle f)$,
- $\langle x \leftarrow t\rangle f \Leftrightarrow f[x/t]$,
- $f \Leftrightarrow \widehat{f}$; *where* $\widehat{f}$ *is* f *in which some occurrence of program* P *has been replaced by the program* $z := x; P'; x := z$, *for* z *not appearing in* f, *and* P' *is* P *with all the occurrences of* x *replaced by* z,

and the inference rules are those used for classical first-order logic and

- *Generalization rule for the* necessary *modal statement:*

$$\frac{f}{[P]f}$$

- *Infinitary convergence rule:*

$$\frac{(\forall n : nat)(f \Rightarrow [P^n]g)}{f \Rightarrow [P^*]g}$$

Once fork algebras and FODLwA have been fully introduced we can define $\mathbf{A_g}$ *signatures* as FODLwA signatures that use the fork algebra constants and functions, and $\mathbf{A_g}$ *theories* as $\mathbf{A_g}$ *signatures* plus the deductive system for both fork algebra and FODL.

In [FPB02], the following two theorems were presented. They are the main reason why we believe $\mathbf{A_g}$ is a good choice as a software modeling language.

The first one shows how the existing gap between the syntax and semantics can be bridged as a consequence of the representability theorem of the fork algebras. It is important because, given a specification, the reader is allowed to reason about it in terms of binary relations.

Theorem 2.4 *Given an* $\mathbf{A_g}$ theory Ψ*, for each model* $\mathfrak{A} = \langle A, m_{\mathfrak{A}} \rangle$ *for* Ψ *there exists a model* $\mathfrak{B} = \langle B, m_{\mathfrak{B}} \rangle$ *for* Ψ *in which the domain* B *consists of concrete binary relations.*

As $\mathbf{A_g}$ is first-order dynamic logic of fork algebra terms, formulas are first-order dynamic logic formulas in which the only predicate symbol appearing is "=". That is the reason the system can be divided in two parts, the first one is the set of axioms presented in 2.2 which allows to prove fork algebra equations. The second part consists of the axioms and inference rules presented in 2.10, which allows to prove the first-order dynamic logic formulas that state properties of the terms involved in the specification. The completeness of the deductive system allows to use theorem provers as syntactic tools for reasoning about specifications, and consequently mechanize the task of system verification.

The last theorem shows the completeness of the deductive system for $\mathbf{A_g}$.

Theorem 2.5 *Completeness of the deductive system of* $\mathbf{A_g}$*. Let* $\Psi \cup \{\psi\}$ *be a set of* $\mathbf{A_g}$ *formulas. Then,*

$$\Psi \vdash_{Ag} \psi \textit{ iff } \Psi \models_{Ag} \psi \ .$$

Chapter 3

The *PVS* Specification and Verification System

PVS (Prototype Verification System) is intended as an environment for constructing clear and precise specifications and for developing readable proofs that have been mechanically verified [ORS92, ORSvH95, Sha01]. A variety of examples have been verified using PVS [CLM+95, ORSSC98]. The most substantial use of PVS has been in the verification of the microcode for selected instructions of a commercial-scale microprocessor called AAMP5 designed by Rockwell-Collins [MS95] and in the LOOP project [vdBJ01]. The key elements of the PVS design are captured by the combination of features listed below.

An expressive language with powerful deductive capabilities. The PVS specification language is based on classical, simply typed, higher-order logic with base types such as the Booleans `bool` and the natural numbers `nat`, and type constructors for functions `[A -> B]`, records `[# a : A, b : B #]`, and tuples `[A, B, C]`. The PVS type system also admits predicate subtypes, e.g., $\{$`i : nat | i > 0`$\}$ is the subtype of positive numbers. The PVS type system includes dependent function, record, and tuple types, e.g., `[# size: nat, elems: [below[size] -> nat] #]` is a dependent record where the type of the `elems` component depends on the value of the `size` component. It is also possible to define recursive abstract datatypes such as lists and trees. The definition of a recursive datatype can be illustrated with the list type of the PVS prelude built from the constructors `cons` and `null`. Theories containing the relevant axioms, induction schemes, and useful datatype operations are generated from the datatype declaration.

```
list [T: TYPE]: DATATYPE
 BEGIN
  null: null?
  cons (car: T, cdr:list):cons?
 END list
```

PVS also has parametric theories, so that it is possible to capture, say, the notion of sorting with respect to arbitrary array sizes, types, and ordering relations. Constraints on the theory parameters can be stated by means of assumptions within the theory. When an instance of a theory is imported with concrete parameters, there are proof obligations for the corresponding instances of the parameter assumptions. A theory is a list of declarations of constants (with or without definitions) and theorems. The PVS typechecker checks a theory for simple type correctness and generates proof obligations (called TCCs for *type correctness conditions*) corresponding to predicate subtypes. Typechecking is undecidable for PVS to the extent that it involves discharging such proof obligations.

Powerful decision procedures with user interaction. PVS proofs are constructed interactively. The primitive inference steps for constructing proofs are quite powerful. They make extensive use of efficient decision procedures for equality and linear arithmetic [Sho84, RS02, SR02]. They also exploit the tight integration between rewriting, the decision procedures, and the use of type information [CRSS94]. PVS also uses BDD-based propositional simplification so that it can combine the capability of simplifying very large propositional expressions with equality, arithmetic, induction, and rewriting.

Higher-level inference steps can be defined by means of strategies (akin to LCF tactics [GMW79]) written in a simple strategy language. Typical strategies include heuristic instantiation of quantifiers, propositional and arithmetic simplification, and induction and rewriting. The PVS proof checker tries to strike a careful balance between an automatic theorem prover and a low-level proof checker.

Model checking with theorem proving. Many forms of finite-state verification, such as linear temporal logic model checking, language containment, and bisimulation checking, can be expressed in the mu-calculus [BCM$^+$90, EL86]. The higher-order logic of PVS is used to define the least and greatest fixpoint operators of the mu-calculus. When the state type is finite, the mu-calculus expressions are translated into the propositional mu-calculus and a propositional mu-calculus model checker can be used as a decision procedure. The finite types are those built from booleans and scalars using records, tuples, or arrays over subranges. Fairness cannot be expressed in CTL, but it can be defined using the mu-calculus. BDD-based symbolic model checking is integrated into PVS as a decision procedure for the Boolean fragment of the mu-calculus [RSS95]. The model checker can be invoked as an interactive proof step together with rewriting, induction, and simplification using the ground decision procedures. Automatic predicate abstraction has been implemented as an interactive step for constructing finite-state property-preserving abstractions of infinite-state systems [SS99]. Though this exercise does not use the model checker and abstractor, these should play an important role in future work.

Deduction and Execution. A functional fragment of PVS has been given an execution semantics that is supported by a code generator which produces Common Lisp code. The code generator includes a destructive update optimization that translates PVS array updates into destructive updates in Common Lisp, when it is safe to do so [Sha02]. The generated code is also safe with respect to runtime errors if it is generated from a typechecked PVS expression where all the generated proof obligations have been discharged.

Chapter 4

Encoding $\mathbf{A_g}$ Semantics in *PVS*

To build a proof checker for $\mathbf{A_g}$ using *PVS* as a semantic framework, we need to encode the semantics of the language within the higher-order logic of *PVS*, and make use of some useful and powerful features such as the abstract data-type mechanism [OS93]. The first step is the construction of the FODLwA language objects in a way that allows us to define their semantics as we did in Section 2.2. Next is the definition of the fork algebra language and the encoding of the point-dense simple proper fork algebra terms as was shown in Section 2.1. Finally, we show how the results of the previous steps must be combined in order to prove properties of $\mathbf{A_g}$ specifications.

4.1 Encoding of FODLwA semantics

The syntax of FODLwA has been formally introduced in Definition 2.6. What we do is to express the language objects in a way that allows us to define their semantics. Due to the inductive nature of the definition of the language, we define it using the abstract datatype mechanism [OS93]. Figure 4.1 contains the definition of the language.

When the specification is typechecked the abstract datatype mechanism generates a new specification file. This new specification contains the theory of the objects defined by the datatype, including the subtypes, map function, and the recursion combinator `reduce_nat`. The recursion combinator will be particularly useful in the encoding of the semantics because it will be the basis for the complexity measure that we need to define the meaning function recursively.

The definition of the language is parametric in the constant, metavariable, variable, predicate and function symbols, and also in the functions that define the arity of the predicate and function symbols. This allows us to work with only one formal language, but using different instances of it depending on the sets of symbols used in particular problems.

Note that this definition of the language is not quite right. There is a restriction that must be satisfied by the FODLwA language objects to be well formed; this restriction establishes that no metavariable symbol is allowed to occur in a program. It is easy to construct a counterexample of this property following the previous definition. It is also easy to define a recursive function that tests if an object is well formed.

```
FODL_Language[Constant: TYPE,
             Metavariable: TYPE,
             Variable: TYPE,
             Predicate: TYPE, sigPredicate: [Predicate -> nat],
             Function_: TYPE, sigFunction_: [Function_ -> nat]]:
       DATATYPE WITH SUBTYPES Term_, Formula_, Program_

  BEGIN

    c(c: Constant): c?: Term_
    m(m: Metavariable): m?: Term_
    v(v: Variable): v?: Term_
    F(f: Function_, lF:
      lPrime: list[Term_] | sigFunction_(f) = length(lPrime)):
           F?: Term_
    TRUE: TRUE?: Formula_
    NOT(f: Formula_): NOT?: Formula_
    IMPLIES(f_0, f_1: Formula_): IMPLIES?: Formula_
    P(p: Predicate, lP:
      lPrime: list[Term_] | sigPredicate(p) = length(lPrime)):
           P?: Formula_
    EXISTS_(x: (v?), f: Formula_): EXISTS?: Formula_
    <>(P: Program_, f: Formula_): DIAMOND?: Formula_
    T?(f: Formula_): T??: Program_
    A(pre_post: [Formula_, Formula_]): A?: Program_
    SKIP: SKIP?: Program_
    <|(x: (v?), t: Term_): ASSIGNMENT?: Program_
    //(P_0, P_1: Program_): COMPOSITION?: Program_
    +(P_0, P_1: Program_): CHOICE?: Program_
    *(P: Program_): ITERATION?: Program_

  END FODL_Language
```

Figure 4.1: Definition of FODLwA language within *PVS*.

Once the function that assures the well-formedness property is defined, we will work only with the objects that satisfy this property. In *PVS* this type is specified as a predicate subtype, for example, we define the type of well-formed terms in the following way: "`wf_Term_: TYPE = {t: Term_ | wf (t)}`", and similarly for formulas and programs.

The semantics of terms, formulas and programs is encoded within *PVS* in the same way as defined in Section 2.2. In Figure 4.2 we show the important parts of the theory that define the meaning function for FODLwA.

The most important fact about this theory is that it introduces not only all the arguments needed to build an instance of the well-formed language; it also is parametric on a set of type predicates. For simplicity this feature was not presented in Section 2 but it does allow for multi-sorted logics. Together with the type predicates we add a pair of functions that map the variable and metavariable symbols to these predicates. The meaning function is defined under the name of `meaningF`, and uses two auxiliary functions, `m` and `mTerm`.

The function `meaningF` is defined in a way that hides the universal quantification over the assignment of values to metavariables, but the functions that really define the semantics of a FODLwA formula are `m`, in the case of formulas and programs, and `mTerm`, in the case of terms.

Note that there are other parameters of this theory. The parameters `mTypePred`, `mConstant`, `mPredicate`, and `mFunction_` are functions that map every type predicate, constant, predicate and function symbol to a real object.

In Figure 4.3 we show the meaning function for terms. The interesting parts of this

```
FODL_semantic[Constant: TYPE,
             Metavariable: TYPE,
             Variable: TYPE,
             Predicate: TYPE, sigPredicate: [Predicate -> nat],
             Function_: TYPE, sigFunction_: [Function_ -> nat],
             TypePred: TYPE,
             TPMetavariable: [Metavariable -> TypePred],
             TPVariable: [Variable -> TypePred],
             Carrier: TYPE+,
             mTypePred: [TypePred -> [Carrier -> bool]],
             mConstant: [Constant -> Carrier],
             mPredicate: [P: Predicate ->
                            [{l: list[Carrier] | sigPredicate(P) = length(l)} ->
                               bool]],
             mFunction_: [F: Function_ ->
                            [{l: list[Carrier] | sigFunction_(F) = length(l)} ->
                               Carrier]]: THEORY

  BEGIN

    ASSUMING
      non_empty_types: ASSUMPTION
         FORALL (t: TypePred): EXISTS (c: (mTypePred(t))): TRUE
    ENDASSUMING

    IMPORTING wf_FODL_Language[Constant,
                               Metavariable,
                               Variable,
                               Predicate, sigPredicate,
                               Function_, sigFunction_],
              list_max

    World_: TYPE = [v: Variable -> (mTypePred(TPVariable(v)))]
    ⋮
    AssMetavariable: TYPE = [m: Metavariable -> (mTypePred(TPMetavariable(m)))]
    ⋮
    mTerm(mMetavariable: AssMetavariable, w: World_)(t: wf_Term_):
        RECURSIVE Carrier = ...

    m(mMetavariable: AssMetavariable)(l: union[wf_Formula_, wf_Program_]):
        RECURSIVE {u: union[PRED[World_], PRED[[World_, World_]]] |
           CASES l OF inl(f): inl?(u), inr(P): inr?(u) ENDCASES} = ...

    meaningF(f: wf_Formula_): PRED[World_] =
      {w: World_ | FORALL (mMetavariable: AssMetavariable):
              left(m(mMetavariable)(inl(f)))(w)}

  END FODL_semantic
```

Figure 4.2: Theory that encodes the semantics of FODLwA.

function are how the previously defined meaning function for constant symbols is used to complete the evaluation of terms in one of the base cases and how the map function of lists is used to compute the meaning of a function symbol application to a list of terms.

The readers interested in the complete encoding of the semantics can take a look at the *PVS* files.[1] Here we just highlight the more interesting encodings. Figure 4.4 shows the use of the type predicates to restrict the universe over which the variable symbols range when they are existentially quantified. In the case of Figure 4.5 the μ-calculus theory, defined

[1] The files are available at `ftp://ftp.csl.sri.com/pub/pvs/examples/AgExample/`.

```
mTerm(mMetavariable: AssMetavariable, w: World_)(t: wf_Term_):
    RECURSIVE Carrier =
  CASES t
    OF c(c_var): mConstant(c_var),
       m(m_var): mMetavariable(m_var),
       v(v_var): w(v_var),
       F(f_var, list_var):
         mFunction_(f_var)(map(mTerm(mMetavariable, w))(list_var))
    ENDCASES
  MEASURE complexity(t)
```

Figure 4.3: Meaning function for terms.

in the *PVS* prelude, is used to compute the meaning of program iteration as a greatest fix point.

```
m(mMetavariable: AssMetavariable)(l: union[wf_Formula_, wf_Program_]):
    RECURSIVE {u: union[PRED[World_], PRED[[World_, World_]]] |
       CASES l OF inl(f): inl?(u), inr(P): inr?(u) ENDCASES} =
  CASES l
    OF inl(f):
         CASES f
           OF
              ⋮
              EXISTS_(var_var, f_var):
                inl({w: World_ |
                       EXISTS (t: (mTypePred(TPVariable(v(var_var))))):
                         left(m(mMetavariable)(inl(f_var)))
                            (w WITH [(v(var_var)) := t])}),
              ⋮
           ENDCASES,
       inr(P):
         CASES P
           OF
              ⋮
           ENDCASES
    ENDCASES
  MEASURE complexity(CASES l OF inl(f): f, inr(P): P ENDCASES)
```

Figure 4.4: Meaning function for the existential quantifier.

4.2 Encoding of the Point-Dense Simple Proper Fork Algebras

As we did in Section 4.1, when we encoded the semantics of FODLwA, we will separate the job of encoding the semantics of the fork calculus in two basic steps; the first one will be the definition of the symbols used to build terms and predicates over them, and the second will be the encoding of the semantics of these symbols.

The definition of the symbols is straightforward, it is defined in terms of finite sets by enumerating the elements they contain. Figure 4.6 shows the interesting parts of this file.

Note that we also defined the function that returns the arity of each of the predicate and function symbols, and that this is the place to declare variable and metavariable symbols

```
m(mMetavariable: AssMetavariable)(l: union[wf_Formula_, wf_Program_]):
    RECURSIVE {u: union[PRED[World_], PRED[[World_, World_]]] |
       CASES l OF inl(f): inl?(u), inr(P): inr?(u) ENDCASES} =
  CASES l
    OF inl(f):
         CASES f
           OF
              .
              .
              .
           ENDCASES,
       inr(P):
         CASES P
           OF
              .
              .
              .
              *(P_var): inr({w: [World_, World_] | (LAMBDA (x: [World_, World_]):
                           mu[[World_, World_]] (LAMBDA (p: PRED[[World_, World_]]):
                              {wp: [World_, World_] |
                                ({wpPrime: [World_, World_] | wpPrime`1 = wpPrime`2})(wp)
                                  OR
                                ({wpPrime: [World_, World_] |
                                    EXISTS (w: World_):
                                        right(m(mMetavariable)(inr(P_var)))(wpPrime`1, w)
                                         AND
                                        p(w, wpPrime`2)})(wp)})
                         (x))(w)})
               .
               .
               .
           ENDCASES
    ENDCASES
  MEASURE complexity(CASES l OF inl(f): f, inr(P): P ENDCASES)
```

Figure 4.5: Meaning function for the iteration program.

for a particular specification.

Once the language for the fork calculus has been declared we need to define its semantics. The semantics of fork calculus terms were introduced in Definitions 2.4 and 2.5. We will now see how to define this in *PVS*.

We first construct the structured universe which interprets fork calculus terms as binary relations. This is presented in Figure 4.7, which defines a binary tree structure over elements. By defining the set of elements in this way, we obtain the closure of the set `T` under the function `pair`. Again, as we mentioned before, the *PVS* type-checking mechanism generates a specification containing the theory of these objects and we define the semantics by importing the theory with appropriate actual parameters.

Note that the abstract data type mechanism generates inductive types, so the universe is finitely generated. This restriction means that this construction represents only a subclass of simple proper fork algebras; in particular, point-dense fork algebras cannot be represented. Fixing this requires removing all inductive declarations from the generated theories, which would lead to the theory presented in Figure 4.8.[2]

In Figures 4.9 and 4.10 we show how point-dense simple proper fork algebra terms are defined using functions (`mConstant`, `mPredicate` and `mFunction_`) that map constant, predicate and function symbols to real constants, predicates and functions defined over

[2]The generated file has advantages aside from providing induction; in particular, the inclusive and disjoint axioms are automatically discharged by the prover. For expediency the actual specifications include the generated files, and we were careful not to make use of induction. We plan to correct this for the next application of this semantics.

```
FA_Language: THEORY

  BEGIN

    Constant: TYPE = {zero, one, one_prime, C_0, C_1, inl, inr,
                      Pi_1, Pi_2, ...}
    ⋮
    Predicate: TYPE = {Leq, Functional, OneToOne, Pair}

    sigPredicate: [Predicate -> nat] =
            LAMBDA (P: Predicate): CASES P
                                   OF Leq: 2,
                                      Functional: 1,
                                      OneToOne: 1,
                                      Pair: 1
                                    ENDCASES

    Function_: TYPE = {sum, product, complement, composition, converse, fork,
                       ...}

    sigFunction_: [Function_ -> nat] =
            LAMBDA (F: Function_): CASES F
                                   OF sum: 2,
                                      product: 2,
                                      complement: 1,
                                      composition: 2,
                                      converse: 1,
                                      fork: 2,
                                      Dom: 1,
                                      FunctionUpdate: 2,
                                      FunctionUndef: 2,
                                      Neg: 1,
                                      ⋮
                                   ENDCASES

  END FA_Language
```

Figure 4.6: Definition of fork calculus language within *PVS*.

```
FA_Element[T: TYPE]: DATATYPE
  BEGIN
    element(t: T): element?
    pair(el0, el1: FA_Element): pair?
  END FA_Element
```

Figure 4.7: Structured universe within *PVS*.

binary relations. In Figure 4.9 we show how the theory that represents the structured universe is imported to build the carrier of the algebra. Notice that we did not use the axioms presented in Section 2 to assure point-density and simplicity, in their place we used an axiom stating that for all a, b elements of the base set of the algebra the relation $\{\langle a, b\rangle\}$ is a member of the carrier; the correctness of this choice follows from [Mad91, Theorem 51].

Finally in Figure 4.10 we show how this carrier is used to give semantics to the symbols.

```
FA_Element_adt[T: TYPE]: THEORY
 BEGIN

  FA_Element: TYPE

  element?, pair?: [FA_Element -> boolean]

  element: [T -> (element?)]

  pair: [[FA_Element, FA_Element] -> (pair?)]

  t: [(element?) -> T]

  el0: [(pair?) -> FA_Element]

  el1: [(pair?) -> FA_Element]

  ord(x: FA_Element): upto(1) = ...

  FA_Element_element_extensionality: AXIOM ...
  FA_Element_element_eta: AXIOM ...
  FA_Element_pair_extensionality: AXIOM ...
  FA_Element_pair_eta: AXIOM ...
  FA_Element_t_element: AXIOM ...
  FA_Element_el0_pair: AXIOM ...
  FA_Element_el1_pair: AXIOM ...
  FA_Element_inclusive: AXIOM ...
  FA_Element_disjoint: AXIOM ...

 END FA_Element_adt
```

Figure 4.8: Theory that models the structured universe within *PVS*.

4.3 Encoding of $\mathbf{A_g}$

Sections 4.1 and 4.2 suffice to build a framework in which it is possible to prove properties written in $\mathbf{A_g}$. This is achieved by building a new theory in which the FODLwA semantics is imported, instantiated on the sets of symbols defined as the language of the fork algebra and the meaning functions for these symbols.

Note that the semantics presented so far just uses standard PVS features and provides the means to prove every property written in the first-order dynamic logic of fork algebras. But we want more than just the basic capability; to be useful the language must allow the introduction of new types, predicates and functions, and in this case atomic actions, that are application specific.

The introduction of new definitions is straightforward. To add a constant, predicate or function definition, simply define its name as an element of the corresponding set in the theory that defines the fork algebra language and the semantics associated with that name. Proving properties of this extended language is the same as described earlier in this section. Adding atomic actions definitions does not present any difficulties, simply define them in a new theory that imports the extended language. This is illustrated in the next section where we show how to build and verify a specification within this framework.

```
FA_semantic: THEORY

  BEGIN

    IMPORTING FA_Language
    ⋮
    Element: TYPE+
    c_0: Element
    c_1: Element
    o_Elements: PRED[Element]

    non_empty_o_Elements: AXIOM
             EXISTS (e: (o_Elements)): TRUE

    empty_intersection_of_types: AXIOM
             FORALL (e: Element):
               (e = c_0 AND NOT e = c_1 AND NOT o_Element(e)) OR
               (NOT e = c_0 AND e = c_1 AND NOT o_Element(e)) OR
               (NOT e = c_0 AND NOT e = c_1 AND o_Element(e))

    IMPORTING FA_Element_adt[Element]
    Carrier: TYPE FROM PRED[[FA_Element, FA_Element]]

    carrier_point_dense: AXIOM
       FORALL (a: FA_Element):
          Carrier_pred(LAMBDA (wp: [FA_Element, FA_Element]):
                                                  wp`1 = a AND wp`2 = a)
    ⋮
  END FA_semantic
```

Figure 4.9: Construction of the carrier of the calculus.

```
FA_semantic: THEORY

  BEGIN
    ⋮
    zero: Carrier = LAMBDA (wp: [FA_Element, FA_Element]): FALSE
    one: Carrier = LAMBDA (wp: [FA_Element, FA_Element]): TRUE
    one_prime: Carrier = LAMBDA (wp: [FA_Element, FA_Element]): wp`1 = wp`2
    ⋮
    sum(c0, c1: Carrier): Carrier = LAMBDA (wp: [FA_Element, FA_Element]): c0(wp) OR c1(wp)
    ⋮
    converse(c: Carrier): Carrier = LAMBDA (wp: [FA_Element, FA_Element]): c((wp`2, wp`1))
    fork(c0, c1: Carrier): Carrier = LAMBDA (wp: [FA_Element, (pair?)]):
                                   c0((wp`1, el0(wp`2))) AND c1((wp`1, el1(wp`2)))
    ⋮
    Leq(c0, c1: Carrier): bool = sum(c0, c1) = c1
    Functional(c: Carrier): bool = Leq(composition(converse(c), c), one_prime)
    ⋮
    mConstant: [Constant -> Carrier] =
      LAMBDA (c: Constant): CASES c
                            OF zero: zero,
                               one: one,
                               one_prime: one_prime,
                               ⋮
                            ENDCASES

    mPredicate: [P: Predicate -> [{l: list[Carrier] | sigPredicate(P) = length(l)} -> bool]] =
      LAMBDA (P: Predicate): CASES P
                            OF Leq:
                                  LAMBDA (l: {lPrime: list[Carrier] | length(lPrime) = 2}):
                                    Leq(nth(l, 0), nth(l, 1)),
                                Functional:
                                  LAMBDA (l: {lPrime: list[Carrier] | length(lPrime) = 1}):
                                    Functional(nth(l, 0)),
                                ⋮
                              ENDCASES

    mFunction_: [F: Function_ -> [{l: list[Carrier] | sigFunction_(F) = length(l)} -> Carrier]] =
      LAMBDA (F: Function_): CASES F
                             OF sum:
                                  LAMBDA (l: {lPrime: list[Carrier] | length(lPrime) = 2}):
                                    sum(nth(l, 0), nth(l, 1)),
                                ⋮
                                converse:
                                  LAMBDA (l: {lPrime: list[Carrier] | length(lPrime) = 1}):
                                    converse(nth(l, 0)),
                                fork:
                                  LAMBDA (l: {lPrime: list[Carrier] | length(lPrime) = 2}):
                                    fork(nth(l, 0), nth(l, 1)),
                                ⋮
                             ENDCASES

  END FA_semantic
```

Figure 4.10: Usage of the structured universe to give semantics to the fork calculus constants, predicates and functions.

Chapter 5

Case Study: A Cache System

In this section we consider as a case study a cache system.[1] This example was presented by Daniel Jackson [JSS01] and was used to compare $\mathbf{A_g}$ with Jackson's specification language *Alloy* in [FBP01].

The cache system is built from a *memory* and a *cache*; the system keeps track of the *dirty* addresses to provide a function that *flushes* the content associated with all these addresses by the cache into the memory taking it to a *consistent* state. The functions that the specification provides are:

- DirtyCacheWrite: this action associates an address with possibly new data in the cache system,
- DirtyFlush: this action flushes the data associated with a certain address in the cache into the memory,
- DirtyLoad: this action loads the data associated with a certain address in the memory into the cache, and
- DirtySetFlush: this action has the same effect as *DirtyFlush* but over all the dirty addresses.

A cache system is said to be in a *consistent* state if all the data in the cache is reflected by the data contained in the memory.

The following figures show some parts of the specification of the cache system written in $\mathbf{A_g}$, and how this specification is translated to *PVS* theories to make the verification possible.

Figure 5.1 shows how the domains of the specification are turned into predicates over the carrier of the calculus. This part of the specification must be located in the theory that defines the semantics of the fork calculus. Note how the primitive domains are expressed as partial identities. The definition of the type `DirtyCacheSystem` is analogous to that of type `DirtyCache`.

[1] During the example the prefix *Dirty* will appear frequently. This is because the example was derived from a simpler one that does not involve any administration of the information corresponding to the dirty addresses.

```
Addr: TYPE
Data: TYPE
Memory: TYPE = Addr -> Data
Cache: TYPE = Addr -> Data
Dirty: TYPE = set[Addr]
DirtyCache: TYPE = {dc: Cache +_circ Dirty |
                         Dirty(dc) <= Dom(Cache(dc))}
DirtyCacheSystem: TYPE = {cs: Memory +_circ DirtyCache |
                               Dom(Cache(DirtyCache(cs)) <= Dom(Memory(cs))}
.
.
.

Element: TYPE+
c_0: Element
c_1: Element
Addr_Element: PRED[Element]
Data_Element: PRED[Element]

non_empty_Addr: AXIOM
   EXISTS (e: (Addr_Element)): TRUE
non_empty_Data: AXIOM
   EXISTS (e: (Data_Element)): TRUE
empty_intersection_of_types: AXIOM
   FORALL (e: Element):
     (e = c_0 AND NOT e = c_1 AND NOT Addr_Element(e) AND NOT Data_Element(e)) OR
     (NOT e = c_0 AND e = c_1 AND NOT Addr_Element(e) AND NOT Data_Element(e)) OR
     (NOT e = c_0 AND NOT e = c_1 AND Addr_Element(e) AND NOT Data_Element(e)) OR
     (NOT e = c_0 AND NOT e = c_1 AND NOT Addr_Element(e) AND Data_Element(e))

IMPORTING FA_Element_adt[Element]
Carrier: TYPE = PRED[[FA_Element, FA_Element]]

one_prime_Addr: Carrier = LAMBDA (wp: [FA_Element, FA_Element]): one_prime(wp) AND Addr(wp`1)
one_prime_Data: Carrier = LAMBDA (wp: [FA_Element, FA_Element]): one_prime(wp) AND Data(wp`1)

Addr: PRED[Carrier] = LAMBDA (c: Carrier):
                        Pair(c) AND Leq(c, one_prime_Addr)
Data: PRED[Carrier] = LAMBDA (c: Carrier):
                        Pair(c) AND Leq(c, one_prime_Data)
Memory: PRED[Carrier] = LAMBDA (c: Carrier):
      Leq(c, composition(composition(one_prime_Addr, one), one_prime_Data)) AND
      Functional(c)
Cache: PRED[Carrier] = LAMBDA (c: Carrier):
      Leq(c, composition(composition(one_prime_Addr, one), one_prime_Data)) AND
      Functional(c)
Dirty: PRED[Carrier] = LAMBDA (c: Carrier): Leq(c, one_prime_Addr)
DirtyCache: PRED[Carrier] = LAMBDA (c: Carrier):
       EXISTS (ca: (Cache), d: (Dirty)):
         c = sum(composition(fork(composition(Pi_1, one_prime_C_0),
                                  composition(Pi_2, composition(ca, inl))),
                             Pi_2),
                 composition(fork(composition(Pi_1, one_prime_C_1),
                                  composition(Pi_2, composition(d, inr))),
                             Pi_2)) AND
         Leq(Dirty(c), Dom(Cache(c)))
DirtyCacheSystem: ...
```

Figure 5.1: Specification of the domains of the cache system.

Figure 5.2 shows the usage of PVS lambda expressions to define $\mathbf{A_g}$ atomic actions that can be applied to any variable. In the same way, the predicate that expresses the consistency of a cache system can be written. Figure 5.3 shows the translation of a property that is desirable to hold over these cache systems. This property expresses that given a cache

```
DirtyCacheWrite: Ag Action[cs: DirtyCacheSystem]
cs = cs0 => [DirtyCacheWrite] EXISTS (a: Addr, d: Data):
          Cache(DirtyCache(cs)) =
             FunctionUpdate(Cache(DirtyCache(cs0)), <a, d>) AND
          Dirty(DirtyCache(cs)) = Dirty(DirtyCache(cs0)) U a) AND
          Memory(cs) = Memory(cs0)
 ⋮

pre_DirtyCacheWrite: [(v?) -> wf_Formula_] =
   LAMBDA (v: (v?)): v = m(cs0)
post_DirtyCacheWrite: [(v?) -> wf_Formula_] =
   LAMBDA (v: (v?)):
     EXISTS_(v(addr),
       (EXISTS_(v(data),
          (wf_F(Cache, (:wf_F(DirtyCache, (:v:)):)) =
             wf_F(FunctionUpdate, (:wf_F(Cache, (:wf_F(DirtyCache, (:m(cs0):)):)),
                                 wf_F(composition, (:wf_F(composition, (:v(addr), c(one):)),
                                                     v(data):)):)) AND
           wf_F(Dirty, (:wf_F(DirtyCache, (:v:)):)) =
             wf_F(sum, (:wf_F(Dirty, (:wf_F(DirtyCache, (:m(cs0):)):)), v(addr):)) AND
           wf_F(Memory, (:v:)) = wf_F(Memory, (:m(cs0):))))))
DirtyCacheWrite: [(v?) -> wf_Program_] =
   LAMBDA (v: (v?)): A(pre_DirtyCacheWrite(v), post_DirtyCacheWrite(v))
⋮
```

Figure 5.2: Specification of the atomic actions of the cache system.

system that satisfies the predicate `NonDirtyCache`,[2] every succession of executions of the atomic actions previously described, followed by an execution of the action `DirtySetFlush`, leaves the cache system in a consistent state.

```
Consistency_criteria: THEOREM
FORALL (cs: DirtyCacheSystem): NonDirtyCache(cs) =>
   [(DirtyCacheWrite(cs)+DirtyFlush(cs)+DirtyLoad(cs)+
     DirtySetFlush(cs))*;DirtySetFlush(cs)]DirtyCacheConsistent(cs)

Consistency_criteria: THEOREM
FORALL (w: World_):
   meaningF(FORALL_(v(cs), NonDirtyCache(v(cs)) IMPLIES
              [](*(DirtyCacheWrite(v(cs))+DirtyFlush(v(cs))+
                   DirtyLoad(v(cs))+DirtySetFlush(v(cs)))//DirtySetFlush(v(cs)),
                 DirtyCacheConsistent(v(cs))))))(w)
```

Figure 5.3: Specification of the properties of the cache system.

Figure 5.4 shows the *PVS* proof script of the property stated in figure 5.3.

Some of the proof commands used during the proof are not built in *PVS* commands, but are strategies implemented to make the proof checker easier to use on $\mathbf{A_g}$ specifications. These strategies are not essential for verifying $\mathbf{A_g}$ specifications, but proofs are more elegant compared with the proofs that can be carried without the strategies.

The PVS language allows a direct encoding of $\mathbf{A_g}$, which was compared with *Alloy* in [FBP01]. One of the advantages of the restrictions imposed by *Alloy* is that their analy-

[2] The predicate `NonDirtyCache` is satisfied if and only if the addresses of the cache that were not involved in the operation of writing in the cache system remain consistent with respect to the data contained in the memory of the cache system.

```
;;; Proof for formula SpecProperties.Consistency_criteria
;;; developed with old decision procedures
(""
 (EXPAND-MEANING 1)
 (EXPAND-MEANING 1)
 (EXPAND-MEANING 1)
 (SKOSIMP*)
 (PURIFY-FODL -1)
 (LEMMA "PDL_4_box_form")
 (INST -1 "DirtyCacheConsistent(v(cs))"
  "*(DirtyCacheWrite(v(cs)) + DirtyFlush(v(cs)) + DirtyLoad(v(cs)) + DirtySetFlush(v(cs)))"
  "DirtySetFlush(v(cs))" "w!1 WITH [(cs) := t!1]")
 (EXPAND-MEANING -1)
 (INST -1 "mMetavariable!1")
 (EXPAND-MEANING -1)
 (PROP)
 (HIDE 2 3)
 (EXPAND-MEANING 1)
 (SKOSIMP*)
 (LEMMA "AllStar_preserves_NonDirtyCache")
 (EXPAND "AllStar_preserves_NonDirtyCache" -1)
 (EXPAND-MEANING -1)(EXPAND-MEANING -1)
 (INST -1 "w!1 WITH [(cs) := t!1]")
 (INST -1 "mMetavariable!1")
 (INST -1 "(w!1 WITH [(cs) := t!1])(cs)")
 (EXPAND-MEANING -1)
 (PROP)
 (("1" (EXPAND-MEANING -1)
       (INST -1 "wPrime!1")
       (PROP)
       (PURIFY-FODL -1)
       (LEMMA "DirtySetFlush_leaves_DirtyCacheConsistent")
       (EXPAND "DirtySetFlush_leaves_DirtyCacheConsistent" -1)
       (EXPAND-MEANING -1)
       (EXPAND-MEANING -1)
       (INST -1 "wPrime!1")
       (INST -1 "mMetavariable!1")
       (INST -1 "wPrime!1(cs)")
       (EXPAND-MEANING -1)
       (PROP)
       (("1" (HIDE -2 -3 -4)
             (PURIFY-FODL))
        ("2" (HIDE -2 -3 2)
             (PURIFY-FODL))))
  ("2" (HIDE -1 2)
       (PURIFY-FODL))))
```

Figure 5.4: Proof of the consistency criteria for the cache system.

sis tool is much more automatic. On the other hand, there are specifications and properties that cannot even be expressed in *Alloy*, but which can be stated and proved in this framework. The main thrust of future work will be to provide more automation so that routine verification is automatic, while more difficult properties can still be analyzed by interacting with the prover. We plan to take advantage of the abstractor and model checker of PVS for much of this automation.

Chapter 6

Conclusions

We developed a semantic framework for the $\mathbf{A_g}$ specification language. This framework makes use of theorem proving techniques applicable to this language, which we believe is a powerful language for designing systems. The framework allows the use of *PVS* to develop $\mathbf{A_g}$ specifications and prove properties about them. Strategies have been developed to make the proofs more automatic.

The availability of this framework means that $\mathbf{A_g}$ may now be used on real problems, to see how useful it is in practice. Of course, in the long run $\mathbf{A_g}$ is expected to be used with many tools, including other theorem provers, model checkers, and static analyzers. But PVS provides an easy way to try out different approaches before committing resources.

The $\mathbf{A_g}$ semantic framework also serves as a good case study of a semantic embedding into the higher-order logic of PVS. Because its implementation makes use of many features of *PVS*, it can be used as the basis for a tutorial on the advanced use of this tool.

There is still much that needs to be done to make this a useful tool. Proof construction can be improved by developing strategies that recognize patterns that appear as consequence of the implementation of the framework so the user can concentrate on what needs to be proved instead of dealing with framework details.

The case study presented here is a toy example, and with a larger application it may turn out that more is needed from the system, including better control of the prettyprinter and more automatic strategies. These may need to be application specific, and tools should be provided so they can be defined conveniently.

Reformulating the specification language may improve the use of *PVS* to validate $\mathbf{A_g}$ specifications. Note that we used dynamic logic to describe dynamic properties of a static model described using fork algebras. Following this, if we can formulate the dynamic properties of the system using fork algebra, the logic used over the terms would no longer be needed and consequently the framework would become simpler.

In fact dynamic logic can be translated to fork algebra due to [FBM01], but this translation was not available at the time the specification language was designed.

Even if fork algebras can embed all the other logics that are relevant for reasoning about system correctness, it is still useful to have a dynamic logic over fork algebras to reason explicitly about evolving program behavior. This is because much of the structure is lost, making it harder to analyze the systems.

Dynamic logic may also be used to specify how designs evolve. If we describe little design decisions, such as adding a new type to the model, in terms of atomic actions, the use of the logic allows to state properties about the process of system design, and can be used to formulate and prove design decision invariants. The same argument can be used to argue in favor of the use of any logic over fork algebra, and in this sense it would be useful to build frameworks for every logic that can state interesting properties of the process of system design, as well as the resulting systems.

Appendix A

Displaying Semantically Embedded Logics

In this section we want to discuss some facts that relate to esthetics. The first is how to display the formulas of a sequent. An $\mathbf{A_g}$ formula is an object of a particular theory that has no meaning until it is interpreted using the semantics defined for those objects.

Suppose now that we have the $\mathbf{A_g}$ formula f. f can not be part of a sequent, because it is not boolean. This is the reason why if we want to prove that f is a theorem, we will express that as trying to prove that the formula $\forall(w : World\_) : meaningF(f)(w)$ holds. Note that there is no ambiguity in saying that f holds, because by Theorem 2.5 f is a theorem if and only if f is valid in the semantics we defined.

This problem is very easy to solve, using the *PVS* conversion mechanism we defined the following conversion for the type `wf_FODL_Formula`:

```
meaning_conv: MACRO [wf_Formula_ -> bool] =
   LAMBDA (f: wf_Formula_): FORALL (w: World_): meaningF(f)(w)
```

Once this function is declared to be a conversion, the user of the framework can simply state "`Theorem_1: THEOREM f`" and it will be converted to "`Theorem_1: THEOREM meaning_conv(f)`" which expands to "`Theorem_1: THEOREM FORALL (w: World_): meaningF(f)(w)`". As the function was defined as a `MACRO`, its expansion is done automatically by *PVS*. This means that when the user tries to prove a theorem declared as "`Theorem_1: THEOREM f`", *PVS* automatically shows the following sequent:

```
|-------
{1} FORALL (w: World_): meaningF(f)(w)
```

The other esthetical aspect of this discussion is the fact that there is no need for the meaning function to be shown in the sequent. There is no ambiguity in saying "`FORALL (w: World_): f(w)`" or "`FORALL (w: World_): meaningF(f)(w)`", and analogously in saying "`FORALL (w: World_): FORALL (M: AssMetavariable): left(m(M)(inl(f)))(w)`" or "`FORALL (w: World_): FORALL (M: AssMetavariable): f(M)(w)`". Note here that we are simply arguing that the meaning function does not need to be shown; it was explained at the beginning of this section why the meaning function is so important in the use of *PVS* as a proof checker for $\mathbf{A_g}$.

As a consequence, it is desirable to modify the function that prints the formulas that appear in the sequent in a more friendly way thus avoiding the appearance of the meaning functions. This is the reason that the strategy `expand-meaning` was introduced, (see figure 5.4 in section 5). In this context, given the $\mathbf{A_g}$ formula f *IMPLIES* g, once the quantifiers are skolemized *PVS* shows a sequent as follows.

```
 ⋮
|-------
 ⋮
{n} (f IMPLIES g)(w)
```

```
 ⋮
|-------
 ⋮
{n} (f IMPLIES g)(M)(w)
```

The different arguments depend on the meaning function applied to f *IMPLIES* g. Note that in either case, the only thing we want is to expand the meaning function in order to obtain a sequent with a formula in which the *IMPLIES* operator is a higher-order logic operator instead of an $\mathbf{A_g}$ operator, and the name of the meaning function is unimportant.

This is similar to the way the Duration Calculus is embedded in PVS as described in [SS93], but the implementation there provides a separate parser and prettyprinter. The solution described above is much easier to implement, but less general. For example, *PVS* does not allow `;` to be used as an operator, so `//` is used instead.

Appendix B

Structure of the Framework

In figure B.1 we present the structure of the framework built to reason about $\mathbf{A_g}$ specifications. After that we give a concise description of the files and how they are related.

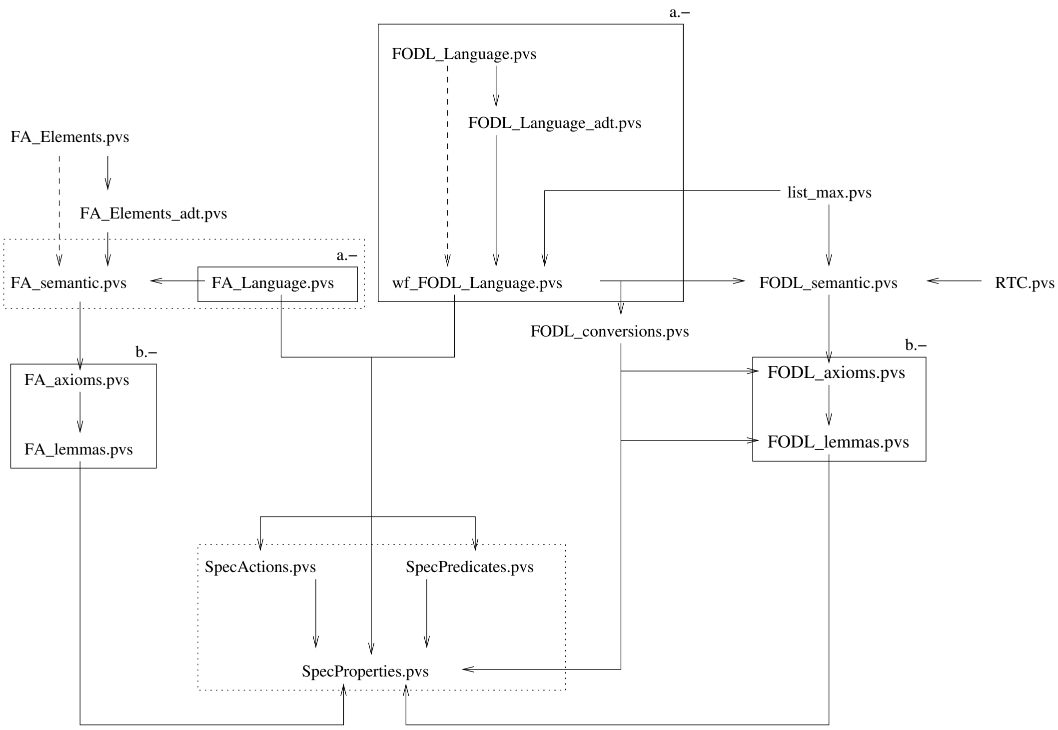

Figure B.1: Structure of the framework.

Solid arrows denote importation, for example, "FODL_semantic.pvs" imports "list_max.pvs". Dashed arrows are used in the cases that the importation occurs through a file generated by the *PVS*' typechecking mechanism, for example, once "FA_Elements.pvs" is typechecked, *PVS* generates the file "FA_Elements_adt.pvs" which contains the theory of the objects defined in "FA_Elements.pvs".

The two solid boxes marked as "a.-" group files that define the syntax of the language; inside one of these boxes appears the file "FA_Language.pvs" that defines the language of the fork calculus, and inside the other there is a group of files ("FODL_Language.pvs", "FODL_Language_adt.pvs" and "wf_FODL_Lan-guage.pvs") that are used to define the language of FODLwA.

Solid boxes marked with a "b.-" group files used to make the framework easier to use; in both cases these files introduce a battery of proved useful lemmas.

Dotted boxes group the files that contain definitions that are specification dependent; the files "FA_Language.pvs" and "FA_semantic.pvs" include the definitions of the domains that appear in the specification, and may introduce new constants, predicates and functions. The files "SpecActions.pvs", "SpecPredicates.pvs" and "SpecProperties.pvs" provide the obvious declarations.

The following is a list of the files that appear in figure B.1 with a brief description of their content:

`FA_Element.pvs`: definition of the elements used to give semantics to the fork calculus,

`FA_Elements_adt.pvs`: theory of the objects declared in the file "FA_-Element.pvs", generated by the *PVS*' typechecking mechanism,

`FA_Language.pvs`: definition of the fork calculus language symbols,

`FA_semantic.pvs`: definition of the fork calculus language semantics,

`FA_axioms.pvs/FA_lemmas.pvs`: commonly used lemmas that can be introduced as part of the proofs,

`FODL_Language.pvs`: definition of the FODLwA language objects,

`FODL_Language_adt.pvs`: theory of the objects declared in the file "FODL_-Language_adt.pvs", generated by the *PVS*' typechecking mechanism,

`wf_FODL_Language.pvs`: definition of the "well-formedness" criteria for the objects of the FODLwA language,

`FODL_semantic.pvs`: definition of the FODLwA language object semantics,

`FODL_axioms.pvs/FODL_lemmas.pvs`: common used lemmas that can be introduced as part of the proofs,

`FODL_conversions.pvs`: definition of some usefull conversions,

`SpecActions.pvs/SpecPredicates.pvs/SpecProperties.pvs`: atomic actions, predicates and properties definitions that come from the specification.

Bibliography

[BCM+90] J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

[BS81] S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Graduate Texts in Mathematics. Springer Verlag, 1981.

[CLM+95] David Cyrluk, Patrick Lincoln, Steven Miller, Paliath Narendran, Sam Owre, Sreeranga Rajan, John Rushby, Natarajan Shankar, Jens Ulrik Skakkebæk, Mandayam Srivas, and Friedrich von Henke. Seven papers on mechanized formal verification. Technical Report SRI-CSL-95-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1995.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 901 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

[dS01] Harrier de Swart, editor. *Proceedings of RelMiCS'6 - TARSKI*, October 2001. Oisterwijk, the Netherlands.

[EL86] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *Proceedings, Symposium on Logic in Computer Science*, pages 267–278, Cambridge, MA, 16–18 June 1986. IEEE Computer Society.

[FBH97] M. F. Frías, G. A. Baum, and A. M. Haeberer. Fork algebras in algebra, logic and computer science. *Fundamenta Informaticae*, 32:1–25, 1997.

[FBHV95] M. F. Frias, G. A. Baum, A. M. Haeberer, and P. A. S. Veloso. Fork algebras are representable. *Bulletin of the Section of Logic*, 24(2):64–75, 1995. University of Lódź.

[FBM01] M. F. Frías, G. A. Baum, and T. S. E. Maibaum. Interpretability of first-order dynamic logic in a relational calculus. In de Swart [dS01], pages 66–80. Oisterwijk, the Netherlands.

[FBP01] Marcelo F. Frías, Gabriel A. Baum, and Carlos G. López Pombo. A comparisson of $\mathbf{A_g}$ with Alloy. In de Swart [dS01], pages 365 – 377. Oisterwijk, the Netherlands.

[FO98] M. F. Frías and E. Orlowska. Equational reasoning in non classical logics. *Journal of Applied Non Classical Logics*, 8(1–2):27–66, 1998.

[FPB02] Marcelo F. Frías, Carlos G. Lopez Pombo, and Gabriel A. Baum. The specification language $\mathbf{A_g}$. Available at `http://www.dc.uba.ar/people/profesores/mfrias/Files/Downloads/Ag.ps`, February 2002.

[Frí02] Marcelo F. Frías. *Fork Algebras in Algebra*. Logic and Computer Science. World Scientific Publishing Co., 2002.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Ham88] A. G. Hamilton. *Logic for mathematicians*. Cambridge University Press, September 1988.

[HKT00] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, October 2000.

[HV91] A. M. Haeberer and P. A. S. Veloso. Partial relations for program derivation: Adequacy, inevitability and expressiveness. In *Working Conference on Constructing Programs from Specifications*, pages 319–371. IFIP TC2, Constructing Programs from Specifications, North Holland, 1991.

[Jac02] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[JSS01] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01)*, Vienna, Austria, 2001.

[Lyn50] R. Lyndon. The representation of relation algebras. *Annals of Mathematics*, 51(2):707–729, 1950.

[Lyn56] R. Lyndon. The representation of relation algebras, part ii. *Annals of Mathematics*, 63(2):294–307, 1956.

[Mad91] Roger D. Maddux. Pair-dense relation algebras. *Transactions of the American Mathematical Society*, 328(1):83–131, November 1991.

[Min00] Paul S. Miner. Analysis of the SPIDER fault-tolerance protocols. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, Hampton, VA, June 2000. NASA Langley Research Center. Slides available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Presentations/lfm2000-spider/`.

[Mor88] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):403–419, 1988.

[MS95] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[ORSSC98] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, October 1998. Springer-Verlag.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[OS93] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report CSL-93-9R, SRI International, December 1993. Subtantially revised in June 1997.

[OSRSC01a] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language reference*. SRI International, version 2.4 edition, December 2001.

[OSRSC01b] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. SRI International, version 2.4 edition, November 2001.

[OSRSC01c] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. SRI International, version 2.4 edition, December 2001.

[Pra76] V.R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 17th Ann. IEEE Symp. on Foundations of Comp. Sci.*, pages 109–121, October 1976.

[RS02] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. Technical report, SRI Computer Science Laboratory, April 2002.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[Sha01] Natarajan Shankar. Using decision procedures with a higher-order logic. In *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages

5–26, Edinburgh, Scotland, September 2001. Springer-Verlag. Available at `ftp://ftp.csl.sri.com/pub/users/shankar/tphols2001.ps.gz`.

[Sha02] Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In A. Pettorossi, editor, *11th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 01)*, volume 2372 of *Lecture Notes in Computer Science*, pages 1–24, Paphos, Cyprus, November 2002. Springer-Verlag. Available at `ftp://ftp.csl.sri.com/pub/users/shankar/lopstr01.pdf`.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SR02] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *International Conference on Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 1–18, Copenhagen, Denmark, July 2002. Springer-Verlag.

[SS93] Jens U. Skakkebæk and N. Shankar. A Duration Calculus proof checker: Using PVS as a semantic framework. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[SS99] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454, Trento, Italy, July 1999. Springer-Verlag.

[Tar41] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6:73–89, 1941.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, Genova, Italy, April 2001. Springer-Verlag.

[VHF95] P. A. S. Veloso, A. M. Haeberer, and M. F. Frías. Fork algebras as algebras of logic. *Abstracts of the Logic Colloquium '94*, page 127, July 1995. Also in Bulletin of Symbolic Logic vol. 1, No. 2 (1995), pp. 265-266.