TLP **14** (4–5): 771–785, 2014. © Cambridge University Press 2014
doi:10.1017/S1471068414000337

A Module System for Domain-Specific Languages

ETHAN K. JACKSON

Research In Software Engineering (RiSE)
Microsoft Research, Redmond, WA, USA 98052
(*e-mail:* ejackson@microsoft.com)

submitted 14 February 2014; accepted 15 March 2014

Abstract

Domain-specific languages (DSLs) are routinely created to simplify difficult or specialized programming tasks. They expose useful abstractions and design patterns in the form of language constructs, provide static semantics to eagerly detect misuse of these constructs, and dynamic semantics to completely define how language constructs interact. However, implementing and composing DSLs is a non-trivial task, and there is a lack of tools and techniques.

We address this problem by presenting a complete module system over LP for DSL construction, reuse, and composition. LP is already useful for DSL design, because it supports executable language specifications using notations familiar to language designers. We extend LP with a module system that is simple (with a few concepts), succinct (for key DSL specification scenarios), and composable (on the level of languages, compilers, and programs). These design choices reflect our use of LP for industrial DSL design. Our module system has been implemented in the FORMULA language, and was used to build key Windows 8 device drivers via DSLs. Though we present our module system as it actually appears in our FORMULA language, our emphasis is on concepts adaptable to other LP languages.

KEYWORDS: module systems, domain-specific languages, logic programming

1 Introduction

Domain-specific languages (DSLs) are routinely created to simplify difficult or specialized programming tasks (Cartey *et al.* 2012; Sangiovanni-Vincentelli *et al.* 2009). They expose useful abstractions and design patterns in the form of language constructs. Unlike *libraries*, which also expose abstractions, DSLs have static semantics to eagerly detect misuse of constructs and dynamic semantics to completely define how language constructs interact (independently of implementation). However, these advantages come with at least two disadvantages: (1) Language design and implementation is challenging, requiring formal specifications, compilers, and debuggers. (2) Composing and reusing DSLs is non-trivial, whereas, on the surface, composing and reusing libraries is as simple as importing them and calling their APIs.

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

Logic programming (LP) is useful for DSL design and implementation (Gurevich 2012; Alvaro *et al.* 2010). Traditionally, programming languages have been specified with a combination of *algebraic data types* (for ASTs) (Hudak 1996), *rules of inference* (for typing and static semantics) (Cardelli 1997), and *abstract transition systems* (for dynamic semantics) (Börger 2005). These specification styles are closely related to logic programs, and rephrasing them as proper logic programs yields formal specifications of DSL semantics that are also language implementations. However, existing LP systems have much less support for composing and reusing DSL specifications. Most systems provide modules for defining and exporting predicates, but this is a low-level form of composition from the perspective of DSLs and compilers (Haemmerlé and Fages 2006).

In this paper we present a complete module system over LP for DSL construction, reuse, and composition. Table 1 gives an overview of our module system and the structure of this paper. It has been designed to be simple (with a few concepts), succinct (for key DSL specification scenarios), and composable (on the level of languages, compilers, and programs). These design choices reflect our use of LP for industrial DSL design. Our module system has been implemented in the FORMULA language (`http://formula.codeplex.com`), and has been used to build DSLs for modeling cyber-physical systems (Simko *et al.* 2013), programming Windows device drivers (Desai *et al.* 2013), and specifying resource allocation problems (Jackson *et al.* 2010), to name a few. Windows 8.0 ships with core components that were built using DSLs specified with FORMULA (e.g. the USB 3.0 stack).

We demonstrate our module system by modularly developing a finite state machine DSL with a small action language. Our examples are as they appear in our FORMULA language, but we emphasize the concepts adaptable to other LP languages. Related work is presented throughout the paper.

2 Domains, Models, and Conformance

Domains specify DSL syntaxes and static semantics, whereas *models* represent DSL programs. There is a *conformance* relationship between domains and models: a model conforms to its domain if it satisfies the domain's static semantics. Domains encapsulate *algebraic data type* (ADT) definitions, rules of inference, and static semantics. Figure 1 shows the *NonDetFSM* domain for a class of non-deterministic finite state machines. Lines 3 – 6 are ADT definitions for the syntax of the DSL. Lines 8 and 9 define the syntax and semantics of the *Reach* judgment, which judges the reachable machine states. Finally, lines 11 – 17 give the static semantics.

2.1 Syntaxes and Algebraic Data Types

ADTs are standard for representing the syntaxes of DSLs and judgments, and our module system makes heavy use of them. Our ADT definitions come in two forms.

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

Table 1. *Overview of module system.*

Modules		
Kind	**Purpose**	**Section**
Domain	Describes DSL syntax, type judgments, and static semantics using *algebraic data types* (ADTs) and LP.	2.1, 2.2
Model	Represents a DSL program w.r.t. a domain as a set of ground facts.	2.3
Transform	A function from models to models, such as a compiler or transition system. Defined using ADTs and LP.	4.2
Language Features		
Feature	**Purpose**	**Section**
Contracts	*Conforms clauses* specify DSL static semantics. *Requires* and *ensures* clauses specify transform behavior.	2.2, 4.4
Symbolic constants	Module-level constants for labeling common sub-expressions and naming value-level transform parameters.	2.3
Inferred rewrites	Compiler-inferred term rewrites that can replace boilerplate recursive rules in transforms.	4.3
Composition Operators		
Feature	**Purpose**	**Section**
Extends, Includes	Compose DSL syntax and static semantics (for domains) and DSL programs (for models); *extends* is conjunctive for static semantics.	3.2, 3.3
Renaming (::)	Generates new modules by systematically renaming data constructors. Used to create product DSLs and define transforms.	4.1
Sequential composition	Creates larger transforms by sequential composition. For instance, a compiler can be decomposed into a sequence of small transforms.	4.4

The first form defines a *data constructor* F:

$$F ::= [\textbf{new}]\ ([arg_1 :]\ T_1, \ldots, [arg_n :]\ T_n).$$

where $[e]$ is an optional expression. This form introduces an n-ary data constructor $F()$ and an identically named type F. The type F denotes all values obtained by applying $F()$ to arguments $a_1, \ldots, a_n$ belonging to types $T_1, \ldots, T_n$. For instance, the *State* type denotes the set of all values constructed by applying *State*() to an integral value. Applying *State*() to a non-integral value creates a badly-typed value. The FORMULA compiler statically guarantees that badly-typed values can never be created (Jackson *et al.* 2012). The arguments of data constructors can optionally be named, e.g. the first argument of *State*() is called *id*. Data constructors marked with the **new** keyword are used for DSL syntax, whereas unmarked constructors are used for the syntax of judgments. Notice that the *Reach*() constructor is unmarked, and can never appear as part of DSL syntax. The second form of type definition

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

```
domain NonDetFSM {
    // FSM Syntax
    State    ::= new (id: Integer).
    Event    ::= new (id: String).
    Trans    ::= new (src: State, ev: Event, dst: State).
    Init     ::= new (st: State).
    // Reachability judgment
    Reach    ::= (State).
    Reach(s) :- Init(s); Reach(s'), Trans(s', _, s).
    // There must be an initial state.
    conforms Init(_).
    // Initial states must be "defined".
    conforms no { i | i is Init, no { s | s is State, s = i.st } }.
    // Transitions must be over "defined" states / events.
    conforms no { t | t is Trans, no { s | s is State, s = t.src } }.
    conforms no { t | t is Trans, no { s | s is State, s = t.dst } }.
    conforms no { t | t is Trans, no { s | s is Event, s = t.ev  } }.
}
```

Fig. 1. (Colour online) Domain module defining a class of non-deterministic finite state machines.

defines a *union type* and has the form: $U ::= T_1 + \ldots + T_n$. Here, U denotes the mathematical union of the types $T_1, \ldots, T_n$.

Related work. There are many variations of ADTs. Our ADTs are a sub-class of *regular types*, with the useful properties that type equality is co-NP-complete and type expressions have canonical forms (Jackson *et al.* 2011). Generalizing to full regular types significantly changes the complexity of static checking, as type equality becomes EXPTIME-complete and types become as expressive as *tree automata* (Aiken and Murphy 1991). For instance, let $s()$ be a unary data constructor standing for the successor function, then regular types can distinguish between even and odd naturals with the productions:

$$Even \rightarrow 0. \quad Even \rightarrow s(Odd). \quad Odd \rightarrow s(Even).$$

Few existing LP systems implement full regular types (e.g. *Ciao* (Hermenegildo *et al.* 2005)). In our experience, a weaker type system is sufficient for capturing syntax, and then more complex properties can be stated directly using LP.

ADTs are also standard in functional programming languages where they are generalized along different dimension compared to regular types (Sulzmann *et al.* 2006). For example, consider these recursive type definitions for two different kinds of lists in the *Haskell* language (Heeren *et al.* 2003):

```
data ListInt = ConsInt Integer ListInt | NilInt
data ListStr = ConsStr String  ListStr | NilStr
```

In Haskell each definition requires a distinct *Nil* constructor (i.e. *NilInt* and *NilStr*), otherwise the type of *Nil* would be both *ListInt* and *ListStr*. In our experience,

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

this restriction is too strong, because it hinders composition of DSL syntaxes. In FORMULA these two definitions can share the same *Nil* constructor as follows:

```
ConsInt ::= (Integer, ListInt). ListInt ::= ConsInt + { Nil }.
ConsStr ::= (String,  ListStr). ListStr ::= ConsStr + { Nil }.
```

(The FORMULA syntax $\{c_1, \ldots, c_n\}$ denotes a set of constants.) Consequently, FORMULA values do not have unique minimal types. This allows syntactic elements to be mixed more freely in composite languages, but comes at the cost of higher-complexity type equality and sub-type testing.

2.2 Judgments and Conformance

LP is used to define rules of inference and static semantics. For example, the reachability judgment (line 9) is a standard recursive rule marking a state as reachable if it can be reached from an initial state by some transitions. To be clear, FORMULA does not support definitions of new *program relations*. Rules examine and populate an implicit unary program relation $p()$, and stratification conditions are more fine-grained to account for this. For instance, if $p()$ were made explicit, then the reachability rule would be written: `p(Reach(s)) :- p(Init(s)); p(Reach(s')), p(Trans(s', _, s)).`

Static semantics are described through *conforms clauses*; such a clause has the form: **conforms** *body*. Where *body* is any expression that could appear on the right-hand side of a standard rule. A DSL program conforms to its domain if every conforms clause is provable, i.e. conforms clauses are conjunctive.

The conforms clause in line 11 requires a FSM to have at least one initial state. The remaining clauses encode a convention that states and events are "defined" by introducing them into the program relation $p()$. Initial states and transitions should only refer to these "defined" values. For instance, it is expected that: $\forall x.\ p(Init(State(x))) \Rightarrow p(State(x))$. The other conforms clauses check these properties using double-negations. Line 13 is read as: *"There should not exist an x where $p(Init(State(x))$ and not $p(State(x))$."* The syntax $\{head \mid body\}$ stands for the set of values produced by the rule *head* :- *body*, and the **no** operator tests if this set is empty. This syntactic sugar allows negations to be nested within a FORMULA rule, though negations must still be stratified. Finally, the syntax x **is** T is shorthand for $p(x), x : T$, meaning x holds in $p()$ and has type T. This syntax allows variables to range over elements of $p()$ based on their type and helps to avoid long patterns of the form $f(_, \ldots, _, x, _, \ldots, _)$.

Related work. A number of LP languages support *integrity constraints* (ICs) through rules of the form *false* :- *body*. A program able to prove the body of an integrity constraint is inconsistent. One could easily envision replacing conforms clauses with ICs. However, there is an important difference. In our approach, the inability to prove conformance is not a logical inconsistency, and this fact allows for more flexible forms of domain composition. The semantics of ICs is rigid and so composing two sets of ICs means all ICs must hold in the composition or the program is logically inconsistent.

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

```
1: model OneStateMach of NonDetFSM
2: {
3:     State(1).
4:     Event("foo").
5:     Init(State(1)).
6:     Trans(State(1), Event("foo"),
7:           State(1)).
8: }
```

```
9: model TwoStateMach of NonDetFSM {
10:     s1 is State(1).
11:     s2 is State(2).
12:     eFoo is Event("foo").
13:     Init(s1).
14:     Trans(s1, eFoo, s2).
15:     Trans(s2, eFoo, s2).
16: }
```

```
17: model BadMach of NonDetFSM { State(1). Init(State(100)). Event("Bar"). }
```

Fig. 2. (Colour online) Several FSM programs encapsulated within models.

2.3 Models

Models define DSL programs. The contents of a model is a set of well-typed ground facts constructed using the new-kind constructors of the model's domain. Semantically, a model is a logic program formed by composing its domain with the set of ground facts enumerated in the model. Figure 2 shows several examples of FSM models. The *OneStateMach* model defines a FSM consisting of a state with ID 1, an event named "Foo", marks 1 as the initial state, and introduces a self-transition on 1 triggered by "Foo". Composing these facts with the *NonDetFSM* domain produces a program where all conforms clauses are provable, and so the model is statically correct. The *BadMach* model does not conform, because the initial state (i.e. $State(100)$) is not an element of $p()$. It violates the conforms clause in line 13 of Figure 1.

The *TwoStateMach* model uses *symbolic constants*, e.g. *s*1 and *eFoo*, to reuse program expressions. Symbolic constants are defined in models using the form: c **is** $F(t_1, \ldots, t_n)$. Such a definition introduces the fact $p(F(t_1, \ldots, t_n))$ and defines a module-level constant c that evaluates to $F(t_1, \ldots, t_n)$. Symbolic constants enable snippets of programs to be shared, sometimes leading to exponentially succinct models. The order of definitions does not matter, though every symbolic constant must be defined and definitions must be acyclic.

3 Domain and Model Composition

3.1 Symbol Tables

Composition is by unioning module definitions. The mechanics of composition depends on symbol definitions and their organization into symbol tables. Table 2 lists the kinds of user-defined symbols along with example introductions. Constructors that may appear in a DSL's syntax are called *new-kind* constructors, whereas those only permitted in judgments are called *derived-kind* constructors. Constants are treated as nullary constructors, and derived-kind constants are introduced by using them on the LHS of rules (and can behave like propositions). A standard idiom for declaring a derived-kind constant q is by the tautology q :- q. Symbols can only have one kind. For instance, in the rule x :- $f(x)$ the symbol x appears both as a variable and a derived-kind constant, which causes a compile-time error. Built-in

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

Table 2. *The kinds of user-introduced symbols.*

Symbol kind	Purpose	Example Introduction
(η) New-kind constructor (arity > 0)	DSL syntax	`F ::= new (...).`
(η) New-kind constructor (arity $= 0$)	DSL syntax	`Nil`, as in `U ::= { Nil }.`
(δ) Derived-kind constructor (arity > 0)	Judgments	`G ::= (...).`
(δ) Dervied-kind constructor (arity $= 0$)	Judgments	`q`, as in `q :- F(x).`
(μ) Union type	Syntax and judgments	`U`, as in `U ::= F + { 1 }.`
(v) Variable	Rules	`x`, as in `q :- F(x).`
(σ) Symbolic constant	Alias model expressions	`c`, as in `c is s(s(0)).`

symbols such as 1 and "foo" are new-kind constants, and built-in data types such as *Integer* and *String* are union types.

Every module has a symbol table recording all definitions. Let $\mathscr{S}$ be the (infinite) set of all possible symbols, $\mathscr{K} \stackrel{def}{=} \{\eta, \delta, \mu, v, \sigma\}$ be the set of symbol kinds, and $\mathscr{T}$ be the set of all terms that can be constructed from $\mathscr{S}$. A symbol table *table* is a partial function from symbols to triples, $table : \mathscr{S} \nrightarrow \mathscr{K} \times \mathscr{N} \times 2^{\mathscr{T}}$. If $table(s) = (k, n, T)$ then symbol s is defined in *table* to have kind k, arity n, and T is the set of well-typed terms that s can represent. Here are some examples:

$$table(1) = (\eta, 0, \{1\}), \quad table(x) = (v, 0, \mathscr{T}), \quad table(Integer) = (\mu, 0, \{\ldots, -1, 0, 1, \ldots\}),$$
$$table(State) = (\eta, 1, \{\ldots, State(-1), State(0), State(1), \ldots\}).$$

A nullary constructor can only construct a single well-typed term. For non-nullary constructors and union types, T is the set of all well-typed terms obeying their type definitions. Variables can be substituted for any term. For symbolic constants, T estimates the evaluation of the constants. The composition of symbol tables $table_1$ and $table_2$ is defined as follows. For each $s \in dom\ table_1 \cup dom\ table_2$:

$$(table_1 \oplus table_2)(s) \mapsto \begin{cases} table_1(s) & \text{if } s \in dom\ table_1 - dom\ table_2, \\ table_2(s) & \text{if } s \in dom\ table_2 - dom\ table_1, \\ (k, n, T) & \text{if } (k, n, T) = table_1(s) = table_2(s), \\ \bot & \text{otherwise.} \end{cases}$$

The composition of tables is legal if no symbol is mapped to $\bot$. This definition allows the same type to be defined in syntactically different ways in several modules. As long as these definitions are semantically equivalent, then the modules can be composed. This flexibility is implemented in FORMULA.

3.1.1 Qualified Symbols and Name Resolution

Returning to the set of symbols $\mathscr{S}$, we add a free associative operator '.' for constructing qualified symbols. If $x, y \in \mathscr{S}$, then $x.y \in \mathscr{S}$ and $(x.y).z = x.(y.z)$. Some examples of qualified symbols are:

(δ) *MyModule.conforms*, (η) *Left.State*, (σ) *MyModel.eFoo*, (η) *In.Left.Trans*

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

Table 3. *Results of lookup() on a table with the symbols: f, A.f, A.A.f, A.B.C.g, A.C.B.g.*

Look-up operation	Result	Explanation
$lookup(table, \epsilon, f)$	f	Symbol with shortest qualifier.
$lookup(table, A, A.f)$	$A.A.f$	Symbol with shortest qualifier.
$lookup(table, A, A.A.f)$	$\perp$	No such symbol.
$lookup(table, \epsilon, B.g)$	$\perp$	Ambiguous, no unique embedding.
$lookup(table, \epsilon, B.C.g)$	$A.B.C.g$	Only compatible symbol.
$lookup(table, B, C.g)$	$\perp$	No such symbol.

Compared to other languages, our module system makes extensive use of qualifiers, so it is important to provide a succinct name resolution strategy. Our strategy is based on *qualifier embeddings*. Let $\vec{q}.s$ be a sequence of atomic symbols $q_1. \ldots .q_n$ qualifying the atomic symbol s. Let ϵ be the empty sequence of qualifiers, e.g. $\epsilon.s = s$. *A sequence* $\vec{p}$ *is embedded in* $\vec{q}$, written $\vec{p} \sqsubseteq \vec{q}$, if $\vec{p} = \epsilon$ or there is a monotone function ι from $\vec{p}$-indices to $\vec{q}$-indices such that $p_1 = q_{\iota(1)}, \ldots, p_{|p|} = q_{\iota(|p|)}$. For example: $b_1.b_2 \sqsubseteq a_1.b_1.a_2.b_2$. The function $lookup(table, \vec{r}, \vec{p}.s)$ returns the shortest symbol from *table* that begins with $\vec{r}$ and embeds $\vec{p}$. Specifically, $lookup(table, \vec{r}, \vec{p}.s) \mapsto \vec{r}\vec{q}.s$ if $\vec{p} \sqsubseteq \vec{q}$ and $\vec{r}\vec{q}.s \in dom\ table$. And, for every other $\vec{q'} \neq \vec{q}$ if $\vec{r}\vec{q'}.s \in dom\ table$ then $|\vec{q'}| > |\vec{q}|$. If no such $\vec{q}$ exists then $lookup(table, \vec{p}.s, \vec{r}) \mapsto \perp$. Our *lookup*() operation searches for symbols in a more general manner than found in other languages. Consequentially, qualified symbols are mostly invisible to the user, even though our composition operators make heavy use of qualifiers. When qualifiers cannot be avoided, the most meaningful qualifiers can be used to disambiguate the symbol. Table 3 illustrates the results of *lookup*() on a sample table.

3.2 Domain Composition

A domain $D \stackrel{def}{=} \langle table, rules \rangle$ consists of a symbol table and set of rules. As a design decision, the derived-kind constants of a domain D are protected by the qualifier D. For instance, the rule q :- q actually introduces a symbol $D.q$ into *table*. Every domain has a constant $D.conforms$ that is provable if and only if all conforms clauses are provable. The ',' operation merges two domains: $D_1, D_2 \stackrel{def}{=} \langle table_1 \oplus table_2, rules_1 \cup rules_2 \rangle$. The result is well-defined if $table_1 \oplus table_2$ does not map a symbol to $\perp$, and if the $rules_1 \cup rules_2$ obey stratification conditions. The *includes* and *extends* operators allow a domain D' to import a set of domains.

domain D' **includes** $D_1, \ldots, D_n$ $\{\ldots\}$. **domain** D' **extends** $D_1, \ldots, D_n$ $\{\ldots\}$.

In both cases, D' is treated as if it contains the merged domain $D_1, \ldots, D_n$. The *extends* operation adds the additional conforms clause to D':

conforms $D_1.conforms, \ldots, D_n.conforms$.

The *extends* operation gives a standard mechanism to conjunct conforms clauses, whereas the *includes* operation allows the user to define their own composite conformance using the derived-kind constants $D_1.conforms, \ldots, D_n.conforms$.

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

```
domain DetFSMWithActions extends NonDetFSM, Actions
{
   ActMap ::= fun (state: State -> actionName: String).
   conforms count({ s | Init(s) }) = 1.
   conforms no { s | Trans(s, e, s'), Trans(s, e, s''), s' != s'' }.
   conforms no { s | s is State, no Reach(s) }.
   conforms no { a | ActMap(_, a), no ActDecl(a, _) }.
}
```

Fig. 3. (Colour online) Deterministic FSMs with an action language defined via domain composition.

```
model CntrActions of Actions
{
  VarDecl("X", INT).
  ActDecl("ZeroX", Asn("X", 0)).
  ActDecl("IncX", Asn("X", BnApp(ADD, Var("X"), 1))).
}
model CntrMach of DetFSMWithActions includes TwoStateMach, CntrActions
{ ActMap(s1, "ZeroX"). ActMap(s2, "IncX"). }
```

Fig. 4. (Colour online) A counter machine formed by model composition.

Figure 3 uses the *merge* and *extends* operators to define a composite DSL of deterministic FSMs with an action language. The action language (see Appendix A) introduces *state variables* and *actions*, which are sequential programs that update state variables. The static semantics of the *Actions* domain includes rules for static type-checking of action bodies. The composite DSL introduces an *ActMap* constructor relating a state to an action that should be executed upon entering that state. The **fun** keyword is like **new**, but requires *ActMap* to be functional (i.e. for all x there is at most one y s.t. $p(ActMap(x, y))$). Finally, the composite language conjoins additional conforms clauses: (1) There is at most one initial state. (2) The transition table of every state is deterministic. (3) Every state is reachable. (4) Every action named in *ActMap* has been declared.

3.3 Model Composition

A model $M \stackrel{def}{=} \langle table, facts \rangle$ is a symbol table and set of facts. M inherits the symbol table of its domain and contains a definition for each symbolic constant c under the fully qualified name $M.c$. Models compose similarly to domains, by composing their symbol tables and unioning fact sets. The *CntrMach* model in Figure 4 describes a counter machine that increments the state variable "X" on every "Foo" event. It is defined by composing the *CntrActions* model with the previous *TwoStateMach* model. *CntrActions* declares an integer variable named "X", an action named "ZeroX" for setting "X" to zero, and an action named

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

```
1: domain ParallelFSMs extends left::DetFSMWithActions, right::DetFSMWithActions
2: {}
3: model ParallelCntrs of ParallelFSMs includes left::CntrMach, right::CntrMach
4: {}
```

Fig. 5. (Colour online) Using renaming to create a DSL for two FSMs running in parallel.

"IncX" for incrementing "X". *CntrMach* assigns state $s1$ to run action "ZeroX" and state $s2$ to run action "IncX".

4 Renaming and Transforms

Transforms are functions from models to models. They are defined with ADTs, strongly-typed logic programs, and a module-level operator called *renaming*.

4.1 Renaming

The renaming operator "::" produces a new module $x :: M$ that is identical to M, except that every occurrence of symbol s in M becomes $x.s$ in $x :: M$. As a design decision, variables and new-kind constants retain their original names. The renaming operator allows the easy construction of distinguished copies of modules. For instance, renaming can be used to construct a DSL representing two distinct FSMs running in parallel. Figure 5 shows how this DSL can be constructed. The *ParallelFSMs* domain is a product domain containing two copies of *DetFSMWithActions* under the renamings *left* and *right*. Symmetrically, the renaming operator can be applied to models. The *ParallelCntrs* model contains two renamed copies of *CntrMach*, thereby creating a valid *ParallelFSMs* model. Appendix B lists the symbol table of this composite model.

4.2 Transforms

Transforms use renaming to label their inputs and outputs. A transform has the shape:

$$\textbf{transform}\ T\ (x_1 :: D_1, \ldots, x_m :: D_m)\ \textbf{returns}\ (y_1 :: E_1, \ldots, y_n :: E_n)\ \{\ldots\}.$$

where D_i and E_j are domains and the labels $x_1, \ldots, x_m, y_1, \ldots, y_n$ are distinct. The transform module T is a composition of its body with the renamed domains in its signature. The signature also indicates that $x_1 :: D_1, \ldots, x_m :: D_m$ are *input domains* and $y_1 :: E_1, \ldots, y_n :: E_n$ are *output domains*.

Figure 6 shows a transform that takes an FSM as input and outputs an equivalent FSM where unreachable states and transitions have been pruned away. Notice the use of renamed constructors to distinguish between input and output values. The first rule (line 5) copies all event declarations from the input to the output. The next rule copies only reachable states. The expression $in.Reach(State(x))$ only requires

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

```
1: transform Prune (in:: NonDetFSM) returns (out:: NonDetFSM)
2: {
3:    requires in.conforms.
4:    ensures out.conforms.
5:    out.Event(n) :- in.Event(n).
6:    out.State(x) :- in.Reach(State(x)).
7:    out.Init(s)  :- in.Init(s).
8:    out.Trans(s, e, s') :- in.Trans(s, e, s'), in.Reach(s), in.Reach(s').
9: }
```

Fig. 6. (Colour online) A transform that prunes dead states.

one qualification, even though *in.Reach*() must have arguments of type *in.State*. This rule is legal because name resolution first uses the qualifier on the outer constructor to resolve the name of an inner constructor (e.g. $lookup(table, in, State)$). If this resolution fails, then it is retried without the outer qualifier.

4.3 Inferred Term Rewrites

Every initial state is by reachable, so the third rule copies all initial states (line 7). This rule looks trivial, but closer inspection reveals a potential problem. Variable s in the RHS must have type *in.State*, but s in the LHS must have type *out.State*. Because these types are disjoint, the compiler should reject this rule as badly typed. Instead, the compiler infers the user's intent to convert values of the form $in.State(a)$ to $out.State(a)$. The rule inferred by the compiler is actually: `out.Init(`$\rho_{in\to out}$`(s)) :- in.Init(s)`. The *relabeling function* $\rho_{in\to out}$ rewrites terms by replacing the *in* qualifier with the *out* qualifier. Relabeling functions are recursive; they rewrite arbitrarily deep terms. In our experience, inferred rewrites are essential for making renaming operator practical.

Due to space limitations we only define the inference problem, but not the requisite algorithms. Let x be a variable occurring in the RHS of a rule and the set T_{rhs} contain (at least) all the values x takes when the body of the rule is satisfied. Also, suppose x occurs somewhere in the LHS and T_{lhs} is the set of all values that x is allowed to take in the LHS. Then x is well-typed if there exists a *unique* relabeling function $\rho_{\vec{p}\to\vec{q}}$ s.t. $\rho_{\vec{p}\to\vec{q}}(T_{rhs}) \subseteq T_{lhs}$. The relabeling of a new-kind constant is itself. Otherwise:

$$\rho_{\vec{p}\to\vec{q}}(\vec{p}\vec{u}.f(t_1,\ldots,t_n)) \stackrel{def}{=} \vec{q}\vec{u}.f(\rho_{\vec{p}\to\vec{q}}(t_1),\ldots,\rho_{\vec{p}\to\vec{q}}(t_n)).$$

Deciding if such a relabeling exists is non-trivial and requires complex type inference algorithms. These have been implemented in FORMULA.

4.4 Execution, Contracts and Composition

A transform T is applied to a sequence of models $M_1,\ldots,M_m$ defined over input domains $D_1,\ldots,D_m$. The mechanics of application are as follows: First, T is composed with the renamed models $x_1 :: M_1,\ldots,x_m :: M_m$ and the resulting logic

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

```
1: transform system PruneAndParallelize (in1:: NonDetFSM, in2:: NonDetFSM)
2: returns (out:: ParallelFSMs)
3: {
4:    prune1 = Prune(in1).
5:    prune2 = Prune(in2).
6:    out    = Parallelize(prune1, prune2).
7: }
```

Fig. 7. (Colour online) Sequential composition of transforms.

program is executed. Next, the j^{th} output model N_j is constructed by collecting all values t such that $p(t)$ holds and $t = y_j\vec{u}.f(\ldots)$. Finally, the renaming y_j is removed from these values yielding an output model purely in the output domain E_j. In symbols:

$$N_j \stackrel{def}{=} \{\ \rho_{y_j \to \epsilon}(t) \mid p(t) \text{ and } t = y_j\vec{u}.f(\ldots) \text{ and } kind(\vec{u}.f) = \eta\}.$$

As usual, only new-kind constructors can appear in output models, hence the extra constraint on the kind of the constructor.

Contracts appear in many programming languages for clearly specifying the intent of methods / functions (Nienaltowski *et al.* 2009; Barnett and Schulte 2003). They can be checked at run-time or compile-time tools can attempt to prove their validity. Our contracts allow users to specify required properties of input models, and properties ensured by the transform when all requirements are met. A clause of the form **requires** *body* states a required property. A clause of the form **ensures** *body* states an ensured property. For example, line 3 of *Prune* requires the input model to conform to its domain. Line 4 guarantees output conformance when the input conforms. As with conforms clauses, transform contracts are conjunctive, and each transform T has derived-kind constants *T.requires* and *T.ensures* that are provable when all requires and ensures clauses are satisfied. A transform contract is a claim that for every application $T(M_1, \ldots, M_m)$ then $p(T.requires) \Rightarrow p(T.ensures)$. Compile-time verification is supported by FORMULA, but outside the scope of this paper.

Finally, transforms can be sequentially composed by listing a series of oriented equations as shown in Figure 7. The *PruneAndParallelize* composite transform takes two FSMs as inputs, prunes the FSMSs, and then combines them into a single *ParallelFSMs* model. The RHS of each equation is a transform application, and the LHS is a sequence of variables that will be bound to the outputs produced by the application. Such a composite is executed by running each constituent transform in dependency-order, while applying renaming and un-renaming at the boundaries of each step. Transform systems can also be sequentially composed within transform systems.

4.5 Related Work

Prolog specifically (Jouault and Bézivin 2006) and LP generally (Horváth *et al.* 2010) have been recognized as useful for *model transformations*. In the former case,

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

Prolog is employed as a behind-the-scenes execution engine, so it was not extended with modules. In the latter case, Prolog-style semantics served as inspiration for the *VIATRA2* language, which is not a LP language. However, model transformation languages such as *VIATRA2* do commonly support various forms of sequential composition (Bisztray *et al.* 2009; Boronat *et al.* 2009), giving further motivation to include it in a module system. LP with constraints has also been used to reason about DSL specifications by translation to LP without extending its module system (Cabot *et al.* 2007).

DSLs have a long history in the software modeling community. The tools and techniques of the modeling community have been heavily influenced by the *Unified Modeling Language* (UML), the *Object Constraint Language*, and the concept of *metamodeling*. However, the resulting amalgam is difficult to formalize (Boronat and Meseguer 2010) and has a complex and underdeveloped module system (Dingel *et al.* 2008). By adding a module system for DSLs directly on top of LP, we inherit the well-understood semantics of LP and avoid problems associated with other notations.

5 Discussion and Future Work

In this paper we presented a complete module system on top of LP for the construction, composition, and reuse of DSLs. Our modules, composition operators, and language extensions were designed to be simple yet synergistic. For instance, inferred term rewrites are a general concept and can also be utilized in all modules. Though our running example illustrated a simple compiler, the same mechanism can also be used to specify the elementary steps of transition systems. This use-case is important for dynamic semantics. Our module system evolved over a period of several years, and was informed by both academic and industrial applications of LP for DSL design. For example, we found a contract language to be essential for documenting the intent of modules in an actionable form.

There are still some important ways in which our module system could be extended. One natural desire is a mechanism to make transforms polymorphic on domains. Another is to package the set of transforms defining an abstract transition system into a single module. The contract language for such a package is also an interesting design problem and would most likely fit well with some form of temporal logic. In our opinion more experimentation and use-cases are required before the right combination of extensions becomes clear. We are currently conducting these experiments while applying FORMULA to industrial examples. The examples presented in this paper along with a binary version of FORMULA can be found at `http://research.microsoft.com/en-us/um/redmond/projects/formula/ICLP2014.html`. The binary version is compiled for Windows machines. The source code for FORMULA and along with its implementation of this module system can be found at `formula.codeplex.com`.

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

References

Aiken, A. and Murphy, B. R. 1991. Implementing Regular Tree Expressions. In *FPCA 1991*. Springer-Verlag, 427–447.

Alvaro, P., Marczak, W. R., Conway, N., Hellerstein, J. M., Maier, D., and Sears, R. 2010. Dedalus: Datalog in Time and Space. In *Datalog*. 262–281.

Barnett, M. and Schulte, W. 2003. Runtime verification of .NET contracts. *Journal of Systems and Software 65,* 3, 199–208.

Bisztray, D., Heckel, R., and Ehrig, H. 2009. Compositionality of Model Transformations. *Electr. Notes Theor. Comput. Sci. 236*, 5–19.

Börger, E. 2005. Abstract State Machines: a unifying view of models of computation and of system design frameworks. *Ann. Pure Appl. Logic 133,* 1-3, 149–171.

Boronat, A., Heckel, R., and Meseguer, J. 2009. Rewriting Logic Semantics and Verification of Model Transformations. In *FASE*. 18–33.

Boronat, A. and Meseguer, J. 2010. An algebraic semantics for MOF. *Formal Asp. Comput. 22,* 3-4, 269–296.

Cabot, J., Clarisó, R., and Riera, D. 2007. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE*. 547–548.

Cardelli, L. 1997. Type Systems. In *The Computer Science and Engineering Handbook*. 2208–2236.

Cartey, L., Lyngsø, R., and de Moor, O. 2012. Synthesising graphics card programs from DSLs. In *PLDI*. 121–132.

Desai, A., Gupta, V., Jackson, E. K., Qadeer, S., Rajamani, S. K., and Zufferey, D. 2013. P: safe asynchronous event-driven programming. In *PLDI*. 321–332.

Dingel, J., Diskin, Z., and Zito, A. 2008. Understanding and improving UML package merge. *Software and System Modeling 7,* 4, 443–467.

Gurevich, Y. 2012. Datalog: A Perspective and the Potential. In *Datalog*. 9–20.

Haemmerlé, R. and Fages, F. 2006. Modules for Prolog Revisited. In *ICLP*. 41–55.

Heeren, B., Leijen, D., and van IJzendoorn, A. 2003. Helium, for learning Haskell. In *Haskell*. 62–71.

Hermenegildo, M. V., Puebla, G., Bueno, F., and López-García, P. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Program. 58,* 1-2, 115–140.

Horváth, Á., Bergmann, G., Ráth, I., and Varró, D. 2010. Experimental assessment of combining pattern matching strategies with viatra2. *STTT 12,* 3-4, 211–230.

Hudak, P. 1996. Building Domain-Specific Embedded Languages. *ACM Computing Surveys 28*.

Jackson, E. K., Bjørner, N., and Schulte, W. 2011. Canonical Regular Types. In *ICLP (Technical Communications)*. 73–83.

Jackson, E. K., Kang, E., Dahlweid, M., Seifert, D., and Santen, T. 2010. Components, platforms and possibilities: towards generic automation for MDA. In *EMSOFT*. 39–48.

Jackson, E. K., Schulte, W., and Bjørner, N. 2012. Detecting Specification Errors in Declarative Languages with Constraints. In *MoDELS*. 399–414.

Jouault, F. and Bézivin, J. 2006. KM3: A DSL for Metamodel Specification. In *FMOODS*. 171–185.

Nienaltowski, P., Meyer, B., and Ostroff, J. S. 2009. Contracts for concurrency. *Formal Asp. Comput. 21,* 4, 305–318.

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press

Sangiovanni-Vincentelli, A. L., Shukla, S. K., Sztipanovits, J., Yang, G., and Mathaikutty, D. 2009. Metamodeling: An Emerging Representation Paradigm for System-Level Design. *IEEE Design & Test of Computers 26,* 3, 54–69.

Simko, G., Lindecker, D., Levendovszky, T., Neema, S., and Sztipanovits, J. 2013. Specification of Cyber-Physical Components with Formal Semantics - Integration and Composition. In *MoDELS*. 471–487.

Sulzmann, M., Wazny, J., and Stuckey, P. J. 2006. A Framework for Extended Algebraic Data Types. In *FLOPS*. 47–64.

https://doi.org/10.1017/S1471068414000337 Published online by Cambridge University Press