Identification of and Solutions to Shortcomings of LCL, a Larch/C Interface Specification Language

Patrice Chalin, Peter Grogono and T. Radhakrishnan

{chalin,grogono,krishnan}@cs.concordia.ca

Concordia University, Department of Computer Science
1455 de Maisonneuve Blvd. West
Montréal, Québec, Canada H3G 1M8

Abstract. We present some of the more significant shortcomings of LCL, a Larch/C specification language used to document the interfaces of modules written in ISO C. We illustrate inadequacies in the definition and insufficiencies in the expressiveness of LCL by means of examples that cover dependencies between objects, the trashing of objects, and implicit parameter constraints in function specifications. A violation of the principle of referential transparency is also shown. We describe changes to the LCL language that overcome the identified shortcomings. Since most of the shortcomings are not particular to LCL, this paper will be of interest to language designers and users of other module interface specification languages.

1 Introduction

The Larch approach to specification promotes the modular development of programs and encourages the use of data abstraction. In Larch there are two specification tiers or levels. The *shared tier* contains specifications (called *traits*) written in the *Larch Shared Language* (LSL) [8]. A trait defines a multisorted first-order theory. The *interface tier* contains interface specifications written in a Larch interface language. There are several Larch interface languages. The most widely used are LCL, LCPP (an interface language for C++) and LM3 (an interface language for Modula-3). Each[1] interface language is specialized for use with a particular programming language. Using constructs and concepts from the programming language, an interface specification describes what resources are being provided by a module.

Specification languages can be used during the entire software development process to document requirements, designs and the interface specifications for modules and program components. One must be careful in choosing an appropriate specification language for the task at hand [9, 1]. The specialization of a specification language to a particular programming language is an important

[1] With the exception of the two generic interface languages GIL [3] and GCIL [14].

characteristic of *module interface specification languages* (MISL's). Most specification languages are general purpose languages. Among the most popular are VDM-SL [13] and Z [18]. These languages are best suited for design specification.

Much less attention has been given to MISL's by the research community than to design or wide-spectrum specification languages. To our knowledge, the only MISL's are the Larch interface languages and an adaptation of the language used with the Trace Assertion Method (TAM) [16]. Of the Larch interface languages, LCL would seem to be the most developed and used. Development of the TAM-based language is at a preliminary stage and a new release is in preparation [10].

MISL's are an excellent way of introducing formal methods into industrial settings [12]. It is particularly important to industry that start-up costs be minimized and that benefits be apparent even with small investments; we believe that MISL's can offer this. Some of the advantages of the use of MISL's are enumerated next[2].

MISL's can be immediately and 'unintrusively' integrated into current industrial development processes [21]. A company that has invested considerable resources in the creation and installation of their development processes (e.g. training of personnel and construction of tools) is more likely to welcome formal methods that can be used in conjunction with inhouse development standards.

One of the greatest challenges faced by industry is the maintenance of legacy code. Not only can MISL's be applied to new developments, they can also be integrated into the maintenance cycle of existing software systems. This is of great value since it means that formal methods can be retroactively brought into projects that were developed without formal methods.

MISL's can be *gradually* integrated into a project:

- they can be applied to isolated portions of a system (such as those aspects for which reliability is most critical),
- MISL's can be used with varying degrees of rigor: from merely documenting function signatures to providing complete behavioral descriptions for functions. In all cases one can reap benefits.

Tool support for most other classes of specification language is limited to type checking. More automated checks can be performed for MISL's. For example, LCLint, a tool for checking LCL specifications and C code, can be used to detect abstraction boundary violations, illicit access to global variables, and undocumented modification of client-visible objects [6]. As another example, Vandevoorde has developed a prototype program optimizer that makes use of the information derived from module interface specifications to perform optimizations that cannot be accomplished by the inspection of code alone [20].

This paper contributes to the evolution of LCL by documenting some of the more significant shortcomings of LCL 2.4[3]and by proposing solutions to the

[2] These advantages are not necessarily exclusive to MISL's—they may be shared by other classes of specification language.

[3] Version 2.4 is the latest public release of LCL.

identified shortcomings. The authoritative references for LCL 2.4 are the Larch book [8] and Tan's PhD thesis [19]. A basic understanding of LCL is assumed.

2 Dependencies Between Objects

In this section we introduce the concept of object dependency and describe how dependencies can arise. We argue that programmers rely on certain "desirable" kinds of dependency and that they tend to overlook other "less desirable" forms. Our examples will serve to illustrate that LCL lacks operations that would allow specifiers to document and reason about dependency relationships in interface specifications.

2.1 Definitions

In C, an object is a region of data storage consisting of a contiguous sequence of storage units [11, p. 2]. In LCL, the term is used in a more abstract sense (in particular because of the need to model objects that are instances of abstract types): an *object* is a container for values of a particular type [8, p. 59].

We say that an object x_1 *depends on* an object x_2 if changing the value contained in x_2 may affect the value contained in x_1. It is possible for x_1 to depend on x_2 without x_2 depending on x_1[4]. If x_1 depends on x_2 or x_2 depends on x_1, then we say that a dependency exists between x_1 and x_2. If x_1 is not dependent on x_2, then we say that x_1 *is independent of* x_2. The objects in a given collection are *independent*, if each object from the collection is independent of every other object in the collection. Given an expression e that refers to an object—e is called an *lvalue* in C—we shall often lighten our prose by speaking of "the object e" instead of the more verbose but precise "the object referred to by e". Thus, for example, we may state that e_1 and e_2 are independent by which we mean that the objects that are denoted by the expressions are independent. As a consequence, we note that if e_1 and e_2 are independent then the expressions cannot be aliases.

Turning to the low-level model of C for an example, we understand that two objects with overlapping regions of storage are dependent on each other. Thus, objects of array, structure and union types depend on the objects that correspond to their members and *vice versa*. For example, given the following declarations

```
struct { int i; } s;
int a[10];
```

`s.i` and `s` depend on each other since these expressions refer to the same region of memory. Also, by definition, `s` and `s.i` are dependent on each other since changing the value of one will affect the value of the other. Similarly, `a`

[4] This kind of asymmetry may exist between instances of an abstract type.

depends on its members—e.g. `a[9]`. On the other hand, `a[0]`, `a[1]`, ..., `a[9]`, and `s.i` are independent. The dependency relationship that holds between an aggregate or union object and its members is one of the kinds of dependency that programmers rely on and actually take for granted.

When dealing with abstract types we can no longer appeal to the low-level concept of overlapping storage for an intuitive model of dependency. Whether a dependency exists between two instances of an abstract type will depend on the implementation of the abstract type [4].

2.2 Motivating Example: Error in the Larch Book

The purpose of this example is twofold: we wish to illustrate that there are legitimate uses of dependencies (beyond those mentioned in Section 2.1) and that there are certain kinds of dependency that are often overlooked by specifiers and implementors.

```
typedef struct {... char name[maxEmployeeName]; ...} employee;

bool employee_setName(employee *e, char na[]) {
   requires nullTerminated(na^);
   modifies e->name;
   ensures result = lenStr(na^) < maxEmployeeName
           ∧ (if result
              then sameStr(e->name', na^)
                   ∧ nullTerminated(e->name')
              else e->name' = e->name^);
   }
```

Fig. 1. An Excerpt from `employee.lcl`

Our example (see Figure 1) is an excerpt from the Larch book **employee** specification [8, p. 65]. This specification is part of a small database program used to store and perform simple queries on employee records. Employee records are represented by the exposed type **employee** which is defined as a C structure. Of the functions provided for manipulating employee records we show only the function **employee_setName**. It can be used to assign a string to the **name** field of an employee record. Before calling **employee_setName**, a client must make sure that the parameter **na** is a null terminated string. The expressions $e^{\wedge}$ and e' denote the values contained in the object referred to by the subexpression e in the pre-state (the program state before function entry) and post-state (the state after function return) respectively. After the call, the name field of the given employee record will be set to the string contained in **na** if the string length is less than **maxEmployeeName**. Otherwise, the name field of the record is left

unchanged. The function result is true if and only if the length of the string contained in **na** is less than **maxEmployeeName**.

Suppose that all of the employee records in a given database begin with either of the titles "Mr." or "Ms." and that the database maintainer wishes to remove the titles. He or she decides to write a program that will accomplish this task by accessing each employee record, say, as the variable **e**, and then performing the call

```
employee_setName(&e,e.name + 3)
```

Unfortunately the program crashes[5] and inspection of the implementation of **employee_setName** reveals the cause:

```
bool employee_setName(employee *e, char na []) {
    int i;

    for (i = 0; na[i] != '0'; i++)
      if (i == maxEmployeeName) return FALSE;
    strcpy(e->name, na);
    return TRUE;
}
```

The particular way in which **employee_setName** is being invoked causes the standard library function **strcpy** to be called with overlapping arguments (since **e->name** and **na** are part of the same array). The behavior of **strcpy** is undefined when it is called under such circumstances [11, §7.11.2.3]. The specification of **employee_setName** does not prohibit calls for which its arguments are dependent. It is possible that the specification inaccurately reflects the intent of its authors or that the source of error is the implementation: in either case the implementation is incorrect with respect to its specification. With appropriate (but small) changes, the implementation can be corrected by making use of the standard library function **memmove** instead of **strcpy** (since **memmove** may be called with overlapping arguments). The reader may wonder whether **memmove** can be specified in LCL; we address this question in Section 2.4.

We can trace the publication of the database program to the original technical report on LCL 1.0 [7]. The program was subsequently revised and published as part of the Larch book [8, §5.3]. To determine the effectiveness of LCLint at detecting certain classes of errors in LCL specifications and their implementations, David Evans applied LCLint to (among others) the database program. Evans writes:

> "The specifications [of the database program] had been checked by the LCL checker [a predecessor of the LCLint tool] ..., and the source code had been compiled and tested extensively. Since the code and specifications were written by experts, and checked copiously by hand prior to

[5] A sample program compiled with gcc version 2.6.3 and run under SunOS release 4.1.3 generates a segmentation fault.

publication, it was expected that not many bugs would be found." [5, p. 41]

The case study "did uncover two abstraction violations, and one legitimate modification error" [5, p. 50]. We have demonstrated an additional error in the database program which has also escaped the scrutiny of the original designers and subsequent reviewers.

This example illustrates that there are legitimate uses of dependencies (such as the dependency permitted between `*e` and `na` in `employee_setName`) beyond those mentioned in Section 2.1. It also illustrates that errors resulting from unexpected dependencies between arguments can easily be overlooked. We believe that this is true because developers have not been encouraged to think about dependencies that may exist among parameters or between parameters and global variables. A specification language that permits dependencies must have constructs that allow the description of dependency relationships as well as a semantic model that supports reasoning about dependencies: LCL is deficient in both these respects.

2.3 Example: *lookup*

The specification given in Figure 2 defines a global `struct` variable `as` consisting of an array of elements, `elts`, and the size of the prefix of `elts` that is in use. It also defines the function `lookup` which can be used to search for an occurrence of the given value `v` in `as`[6]. If `v` is present in `as`, then `*i` is set to the index of an element of `as` containing `v` and `as` is left unchanged; otherwise, `v` is added to `as` and `*i` is set to the index of the newly added value. The function result is true precisely when the value `v` occurs in `as` (before `lookup` is invoked). The predicate that follows the `else` in the ensures clause of `lookup` is not shown since it is not relevant to our discussion.

After a careful review, the reader may feel that the specification of `lookup` is accurate. It is actually inconsistent—there is no implementation that can satisfy it—since there are situations for which the postcondition cannot be satisfied. For example, suppose that `v` occurs in `as` and that `*i` is an alias for `as.size` or any of the elements of `as.elts` that are in use. Then the ensures clause states that the value of `*i` may change while requiring that the value of `as` remain unchanged; this constraint, in general, will be unsatisfiable in the presence of the described aliasing.

We can attempt to remedy the situation by strengthening the precondition of `lookup` so that `*i` is prohibited from being an alias for any of the subcomponents of `as` (see Figure 3). The resulting specification is less clear and more complex (this augments the risk of introducing errors into the specification) and less maintainable since the specification is now more sensitive to changes in the `AS` structure.

[6] We will at times use the term "as" to refer to the prefix of `as.elts` that is in use.

```
constant int N;
struct AS {int size; int elts[N];} as;

bool lookup(int v, int *i) struct AS as; {
  requires as.size^ < N;
  modifies *i, as;
  ensures    result = v ∈ prefix(as.elts^,as.size^)
           ∧ if result
             then  0 ≤ (*i)' ∧ (*i)' < as.size^
               ∧ as.elts^ [(*i)'] = v
               ∧ as' = as^
             else /* v is inserted into as.elts */ ...;
}
```

Fig. 2. Specification of *lookup*

```
bool lookup(int v, int *i) struct AS as; {
  requires as.size^ < N  ∧  *i ≠ as.size
           ∧ (∀ j:int ((0 ≤ j ∧ j ≤ as.size^)
                       ⇒ *i ≠ as.elts[j]));
  ...
}
```

Fig. 3. Strengthened Precondition for *lookup*

More importantly, the specification is still inconsistent since it is possible for `*i` and `as` to satisfy the requires clause without being independent. In formulating the strengthened precondition we have relied on the following *false* assumption: if two distinct objects are instances of base types (`char`, `int`, etc.), then they must be independent. In C, as in some other imperative programming languages, this assumption can be invalidated by the use of union types. Type casting can also invalidate the assumption.

This example illustrates the need for new LCL language constructs which would allow specifiers to accurately and succinctly express the independence of objects.

2.4 Example: ISO C String Library Functions

It would be reasonable to expect LCL to be expressive enough to allow one to document the behavior of most ISO C standard library functions. Consider the task of writing specifications for the standard string copying functions `memcpy` and `memmove` [11, §7.11.2].

```
void *memcpy(void *s1, const void *s2, size_t n);
```

```
void *memmove(void *s1, const void *s2, size_t n);
```

Both functions can be used to copy **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. There is an extra requirement for **memcpy**: the objects ***s1** and ***s2** must not overlap [11, §7.11.2]. It is impossible to write an LCL specification for **memcpy** since we cannot express the requirement that its arguments are independent of each other.

2.5 Dependencies and Abstract Types

The **fresh** operator is the only LCL operator, other than equality over objects, that allows specifiers to document dependency relationships between objects. An occurrence of the expression **fresh**(*e*) in the ensures clause of a function specification asserts that the object referred to by *e* is not aliased to any object that was visible to the client before function entry [8, p. 77]. By means of the next example, we highlight the need for LCL primitives that would allow for a more precise description of the dependency relationships that may exist between objects.

Most abstract type constructors yield instances of the abstract type that are independent of other client-visible objects. It is not uncommon, though, to find "quick" or "destructive" versions of some constructors that fail to guarantee the independence of the resulting abstract type instance; independence is sacrificed for sake of efficiency.

```
mutable type List;

uses List(int,List);

List mkList(void) {
  ensures result' = empty ∧ fresh(result);
}
List concat(List x1, List x2) {
  ensures result' = x1^ || x2^ ∧ fresh(result);
}
List fastConcat(List x1, List x2) {
  ensures result' = x1^ || x2^;
}
```

Fig. 4. List specification.

For example, a list module might provide two versions of the concatenation operation—see Figure 4. Notice that the specification of **fastConcat** does not ensure **fresh(result)**. It would be more useful, for example, if we could assert that the only dependency created by **fastConcat** is between **result** and **x2**. This extra information would allow us to make better use of **fastConcat**, for example, in the optimization of a series of successive concatenations.

3 Implicit Constraints on Parameters

In LCL, the specifications of functions with parameters have implicit constraints, derived from the parameter declarations, that affect the meaning of the specifications. Unfortunately, most of these implicit constraints are either not documented or inadequately defined. The purpose of this section is to expose some of these implicit parameter constraints and to discuss the consequences of their inclusion in LCL.

3.1 Constraint for All Parameters

There is an implicit constraint that applies to all parameters in a function specification. It requires that the parameters be *defined*. This implicit constraint on parameters is not documented in the Larch book [8] nor in Tan's semantics [19]. We have only been able to find an explicit statement of the constraint in Evans's thesis:

> "LCL specifications denote if the values associated with parameters are defined. ... All other parameters [i.e. other than **out**-qualified pointer parameters] are assumed to be defined when the function is entered." [5, p. 36]

(The **out** parameter qualifier is discussed in Section 3.3.) For example, consider the function **empset_clear** from the Larch book **empset** specification [8, p. 73]:

```
void empset_clear(empset s) {
   modifies s;
   ensures s' = { };
   }
```

By the absence of a requires clause, no explicit requirements are placed on clients of **empset_clear**. Implicitly, though, it is assumed that on function entry, **s** is bound to a defined **empset** (as can be concluded from the informal description of **empset_clear**): "**empset_clear**, is provided for reinitializing an existing **empset**" [8, p. 76].

3.2 Parameters of Pointer Types

There is an additional constraint for parameters of pointer types. The implicit property requires that a pointer parameter reference an allocated object and that this object be defined. This constraint is not documented in the Larch book nor in Tan's semantics[7]. Evans writes:

[7] Tan documents the effect of the out parameter qualifier as applied to parameters of pointer types, but he fails to describe the implicit constraints derived from pointer parameters that are *not* qualified with out.

> "Normally, if a parameter to a function is a pointer, it is assumed that the value it points to is defined and may be used in the body of the function." [5, p. 36]

We discuss some of the shortcomings associated with this implicit constraint.

Constraint is Overly Restrictive The implicit constraint for pointer parameters is overly restrictive since it prevents us from using certain useful implementation techniques. Consider the specification fragment

```
typedef struct node { ... } *List;
constant List emptyList = 0;
List mkList(int info, List tail) { ... }
```

in which the empty list is represented by a null pointer. The function **mkList** is meant to allow clients to construct a new list from a given integer and list. The implicit constraint for pointer parameters effectively prohibits us from representing the empty list by means of a null pointer, since, for example, we cannot call **mkList** with **emptyList** as an argument for **tail**. This is because all pointer parameters must refer to allocated objects and a null pointer "is guaranteed to compare unequal to a pointer to any object or function" [11, §6.2.2.3]—i.e., a null pointer can never refer to an allocated object.

Constraint is Ambiguous and Problematic From a given pointer parameter **p** we can access all of the objects **p+i** for **i** in the index set

$$I = \{\, i \mid \texttt{minIndex(p)} \leq i \leq \texttt{maxIndex(p)} \,\}$$

[8, p. 60]. With this in mind, there would seem to be two reasonable interpretations for the implicit constraint. Firstly, we can interpret the implicit constraint as applying to all of the objects that can be accessed via **p**: i.e. all objects **p+i** (for $\texttt{i} \in I$) would have to be allocated and defined. Such an interpretation renders the constraint too restrictive. For example, this would require that every member of a string (represented by a pointer into an array of **char**) be initialized before the string is passed as an argument, even if the string does not occupy the entire array. There is no reason to require that the string be initialized beyond the null character that terminates the string.

Another possible interpretation for the implicit constraint would require that all objects **p+i** ($\texttt{i} \in I$) be allocated but that only the object at **p** need be defined. Assuming $1 \in I$, how would a specifier express the additional requirement that **p+1** be defined? There are no LCL language constructs available to the specifier that would allow the expression of this property.

3.3 The out Parameter Qualifier

It is common in C for a function to return values to its caller by means of objects that are referenced by the function's pointer parameters; the **out** parameter qualifier serves to indicate which parameters are being used for this purpose [19, §4.3]. The specification of **add** given in Figure 5 illustrates the use of **out**. The **out** qualifier has the effect of partly "relaxing" the extra constraint that

```
void add(int m, int n, out int *sum) {
  modifies *sum;
  ensures  (*sum)' = m + n;
}
```

Fig. 5. Use of out in a function specification.

is usually applied to pointer parameters. An **out** qualified pointer parameter is still implicitly required to refer to an allocated object, but that object need not be defined [19, §4.3].

As a final remark, we highlight a contradiction in [19]: although Tan states that the **out** qualifier is applicable *only* to parameters of pointer types [19, §4.3], he also applies it to array parameters [19, §D.27]. Of course, this more liberal use of **out** is reasonable (and is accepted by LCLint), but it has not been documented. Array parameters are discussed in Section 3.4.

3.4 Parameters of Array Types

Although we have found no explicit description of it, there is an implicit constraint on array parameters that is similar to the one for pointer parameters. This would seem reasonable, due to the close relationship between pointers and arrays in C. In fact, someone familiar with C might think that it would be unnecessary to reformulate the implicit constraint for pointer parameters in terms of array parameters because the type of an array parameter is "adjusted to" a pointer type [11, §6.7.1]. In LCL, parameters of array types have a different semantics from those of pointer types [8, p. 60], [19, §7.3.1]—in particular, array parameters are not treated as pointer parameters.

The specification of **date_parse** [19, §D.28] given in Figure 6 provides evidence of the implicit assumption that array parameters refer to objects that have been *allocated* and whose contents are defined. In the specification, **cstring**'s are null-terminated arrays of **char**. If **indate** is a well-formatted date, then this date is parsed and returned in ***d**. The function **date_parse** makes use of the content of **indate**, hence **indate** must refer to allocated storage and its contents must be defined.

The implicit constraint over array parameters suffers from the same ambiguities and drawbacks as the constraint for pointer parameters discussed in

```
bool date_parse (cstring indate,..., out date *d)... {
  modifies ...;
  ensures  result = okDateFormat(getString(indate^))
       ∧ if result
          then (*d)' = string2date(getString(indate^))
          ...;
}
```

Fig. 6. Tan's `date_parse` Function

Section 3.2; i.e., it is not clear whether the implicit constraint requires that all or only some of the array elements be defined—either interpretation leads to difficulties.

3.5 Parameters of Other Types

Consider a function specification with the header

```
void f(int **i)
```

The implicit constraints require that `i` be defined and that `*i` be allocated and defined. Suppose that we further wished to constrain the parameter by requiring that `**i` be allocated and defined. We cannot document this extra property for lack of language primitives in LCL. Similar remarks can be made about parameters of other types (e.g. array of pointer, struct containing a pointer member).

3.6 Parameters *vs.* Global Variables

In designing a module one must decide on the mechanisms by which information will be communicated between the module and its clients. In particular, one must choose between information exchange by means of function parameters or global variables. A designer's freedom of choice is impeded (in favor of the use of function parameters) by the lack of expressiveness of LCL.

For example, given

```
int *gv;

void f(int *pv) { ... }
void g(void) int *gv; { ... }
```

one could not express, in the specification of `g`, a constraint on `gv` that would be equivalent to the implicit parameter constraint on `pv` in `f`. This is because, unlike for function parameters, implicit constraints are not imposed on variables (like `gv`) that are part of the global variable list of a function specification. It is also because there are no language constructs in LCL that express the property that a given object is allocated, or that it is both allocated and defined.

4 Trashing of Objects

The **trashed** operator can be used in the ensures clause of a function specification to indicate that a given object cannot be reliably accessed after the function returns. The **trashed** operator is typically used in the specifications of functions that deallocate memory or that dispose of instances of mutable abstract types. For example, after a call to the function **trashIntObj**

```
void trashIntObj(int *i) {
  modifies *i;
  ensures  trashed(*i);
}
```

a client must not attempt to access the contents of ***i** "because referencing a trashed object can even cause the client program to crash" [8, p. 76]. Notice the presence of ***i** in the modifies clause: an object can be trashed only if it is listed in the modifies clause—although specifications in the LCL literature consistently mention trashed objects in the modifies clause, there is no explicit statement of this requirement. Hence, the modifies clause plays a dual role: it serves to identify those objects that may be trashed as well as those objects that may be preserved but whose values may be modified.

On the other hand, after the invocation of **changeVal**

```
void changeVal(int *i) {
  modifies *i;
  ensures  true;
}
```

a client may still make use of ***i** (though no constraint is placed on the value contained in ***i**) [8, p. 76]. Thus, an object that is not explicitly trashed is implicitly preserved—i.e. *not* trashed. We will illustrate next that this aspect of the semantics of LCL can lead to contradictory interpretations for function specifications that should have the same meaning.

4.1 Referential Opacity

Consider the following specification of **trashOrChange**, which may nondeterministically choose between trashing and not trashing ***i**:

```
void trashOrChange(int *i) {
  modifies *i;
  ensures  trashed(*i) ∨ ¬trashed(*i);
}
```

The predicate in the ensures clause is an instance of the law of excluded middle and hence, it is logically equivalent to **true**. One would expect to be able to simplify the ensures clause while preserving the meaning of the specification.

```
void trashOrChange(int *i) {
  modifies *i;
  ensures  true;
}
```

The resulting specification of **trashOrChange** cannot trash ***i** because of the implicit constraint that ***i** be preserved.

We have illustrated a violation of the principle of referential transparency which states, in essence, that the only important property of an expression is its value and that we can, consequently, substitute equals for equals. Referential transparency is a fundamental principle of mathematical formalisms.

Not only do formal specification languages permit precise documentation, but they also provide the grounds for the formal analysis and transformation of specifications. Formal arguments are most often conducted within a proof system (rather than by direct application of a model theory). For example, in the Refinement Calculus [15], one can make use of "refinement laws" (which can be used as proof rules) to establish the correctness of an implementation with respect to its specification. As a consequence of the identified referential opacity, we note that laws, such as the strengthen postcondition law, do not hold for LCL [2].

5 Shortcomings Resolved

5.1 Dependencies Between Objects

The history of programming languages has been marked by a tendency to make languages more abstract. Increasingly, languages are based on programming concepts (i.e. *semantic objects*) that allow designers to think at a level of abstraction that is closer to the problem domain and further from the computer architectures on which the programs are being executed. In the programming language community, object dependencies tend to be frowned upon. High-level languages tend to severely restrict the kinds of dependency that can be created and low-level languages are characterized by the opposite. In the extreme, object dependencies are prohibited from high-level languages—as in logic or functional programming languages in which computation is based on values rather than objects (by definition, object dependencies cannot exist between values, only between objects). It is important to note that object dependencies *cannot be eliminated* from imperative programming languages that support abstract and indexable[8] types.

By suggesting the systematic adherence to certain programming conventions (e.g. with respect to mechanisms for the implementation and use of abstract types), LCL attempts to raise the level of abstraction at which C programmers think. In providing a semantics for LCL, there would seem to be a tension: although use of LCL promotes C programming at a higher level of abstraction, it is also necessary that the semantic model of LCL subsume that of C since

[8] E.g. array or dynamic types.

LCL is an interface specification language *for* C. The LCL semantic model must capture the behavior of as large a class of C programs as is possible. Hence arises the question: to what degree should dependencies be supported in LCL?

Usually, a model that supports descriptions from two levels of abstraction must be defined in terms of concepts that are from the lowest level. Hence, the semantic model for LCL must accurately capture the kinds of object dependency that can be created in C programs. Our approach to modeling dependencies is formally described in [2]. Of course, it is also necessary that the LCL language have an expressively complete set of constructs for describing dependency relationships. These constructs are introduced next.

In its full generality, the object dependency relation is a dynamic property. For example, dependencies between instances of abstract types implemented by shared realizations may change at run-time [4]. Modeling the object dependency relation as a dynamic property would complicate the semantics and would have important repercussions at the language level. It is not clear, at this point in our research, what language constructs would be best suited to supporting a dynamic dependency relation. The extent to which the dynamic quality of the dependency relation would be actually needed in documenting interface specifications is also unclear. Consequently, in this version of the semantic model the object dependency relation is represented by a static relation, that is, a relation whose value is independent of the program state.

We propose the introduction, in LCL, of two predicates:

- **depOn**(e, e') holds when the object referred to by e depends on[9] the object referred to by e'.
- **indep**$(e_1, e_2, \ldots, e_n)$ holds when the expressions e_1, e_2, ..., e_n denote objects that are independent.

The **depOn** predicate allows specifiers to describe any (static) dependency relation that can exist between objects. Although **indep** can be defined in terms of **depOn**, **indep** is more likely to be used in practice since we generally wish to specify that the objects in a given collection are independent (as opposed to characterizing a particular dependency relationship). For example, **indep** can be used to write concise and accurate specifications for the functions **lookup** and **memcpy**. Concretely, in the case of **lookup**, we capture the requirement that **as** and ***i** be independent by adding **indep(as,*i)** to the requires clause:

```
bool lookup(int v, int *i) struct AS as; {
  requires as.size^ < N ∧ indep(as,*i);
  ...
}
```

The last example of Section 2 required that we be able to strengthen the specification of **fastConcat** by ensuring that the only dependencies created by **fastConcat** are between **result** and **x2**. More precisely, we wish to ensure that

[9] The definition of dependence is given in Section 2.1.

result is independent of any client-visible object that is active in the pre- and post-states and that is also independent of **x2**. One way of rewriting the specification to include this property is as follows[10]

```
List fastConcat(List x1, List x2) {
  ensures ∀ void *x (
            ((*x)\activePre ∧ (*x)\activePost
            ∧ indep(*x,x2)) ⇒
                indep(result,*x))
          ∧ result' = x1^ || x2^;
}
```

(The \activePre and \activePost operators are discussed in the next section.) The ensures clause is somewhat intimidating. Frequent occurrence, in specifications, of properties like these may warrant the introduction of special notation that would allow us to say, e.g. "**fresh(result)** *except for* **x2**."

5.2 Implicit Constraints on Parameters

Constraints for All Parameters The values contained in objects are inevitably encoded in some medium—e.g. volatile storage. It may be the case that for a given object of type T some encodings—e.g. bit patterns—will not correspond to values of type T. We say that an object is *well-defined* with respect to a type T if it contains an encoding that corresponds to a value of type T; that is, if the object contains a *valid representation* of a value of type T. When we say, without qualification, that an object is well-defined, we mean that the object is well-defined with respect to its declared type.

Although the LCL literature is not clear about the logical foundations of LCL, we have chosen LL, the logic underlying LSL to be the logical base for LCL. LL is a first-order multisorted logic with equality in which all function symbols are interpreted as total functions and sorts do *not* have distinguished "undefined" values [2]. Hence, we cannot model undefined *values* in LCL—although we do model non-well-defined objects. The implicit constraint, discussed in Section 3.1, that "all parameters must be defined" becomes a fundamental consequence of the semantic model of LCL and is therefore no longer an implicit constraint.

New LCL Operators In Sections 3.2, 3.5 and 3.6, we noted that it is not possible in LCL to express the property that an object is allocated or that it is both allocated and well-defined. For this purpose we propose the introduction of the following boolean operators

```
__ \activePre,  __ \wellDefPre,
__ \activePost, __ \wellDefPost,
__ \activeAny,  __ \wellDefAny : T → Bool
```

[10] The notation that we are using for the declaration of the quantifier variable is not the notation of LCL 2.4.

The expression e\activePre holds when the object e is active (i.e. allocated) in the pre-state. e\wellDefPre holds when the object e is active and well-defined in the pre-state. The other operations provide similar predicates over the post and generic states. Note that the meaning of the trashed operator can be given in terms of \activePost

```
trashed(gv) ⇔ ¬(gv\activePost)
```

Due to the problems discussed in Section 3, the implicit constraints for pointer and array parameters are dropped. The new operators can be used to express the necessary constraints. For example, the following specification of `f` requires that the object pointed to by `i` be allocated and that the global variable `gv` be well-defined. The function ensures that the post-state value of `*i` is well-defined and that it is equal to the pre-state value of `gv`.

```
void f(int *i) int gv; {
  requires (*i)\activePre ∧ gv\wellDefPre
  modifies *i;
  ensures  (*i)\wellDefPost ∧ (*i)' = gv^;
}
```

Although this approach results in function specifications that are more verbose; elsewhere [2], we have suggested the use of Ada-like parameter qualifiers (`in`, `out`, `inout`) that would allow us to recover the original terseness.

5.3 Trashing of Objects

The semantics of function specifications, in LCL 2.4, is defined in such a way that under certain circumstances some objects are implicitly preserved. We now explain this aspect of the semantics of LCL 2.4 in more detail than in Section 4 and we reexamine the resulting violation of the principle of referential transparency.

The *modified set* of a function specification consists of those objects that are referenced by expressions occurring in the modifies clause. The *trashed set* of a function specification consists of those objects that are referenced by expressions occurring as arguments to the `trashed` operator in the ensures clause [19, §7.4.1]. For example, the modified and trashed sets for the following specification of `trashSome` are {`*a`, `b`, `*c`}, and {`*a`, `b`} respectively.

```
mutable type M;

void trashSome(int *a, M b, int *c) {
  modifies *a,b,*c;
  ensures  (*c)' = (*c)^ + 1 ∧ trashed(b)
           ∧ (if (*a)^ != (*c)^ then
               then ¬trashed(*a) ∧ (*a)' = (*c)^
               else trashed(*a));
}
```

As was indicated in Section 4, an object that is a member of the modified set may be either trashed or modified. An object in the modified set is implicitly preserved only if it is not a member of the trashed set. In the **trashSome** example, *c is implicitly preserved. Thus, the presence or absence of certain argument expressions (of the **trashed** operator) affects the meaning of the function specification. Since the meaning of a function specification depends on more than the truth or falsity of the ensures clause predicate, this clearly leads to a violation of the principle of referential transparency. To recover referential transparency we need only eliminate that aspect of the semantics that relies on the presence or absence of argument expressions to the **trashed** operator.

An obvious approach to achieving this would preserve the dual role of the modifies clause while eliminating the implicit constraint that objects in the trashed set are implictly preserved. As a consequence of this approach specifiers would have to explicitly indicate when objects are to be preserved. For example, the specification of **trashSome** would have to be rewritten as

```
void trashSome(int *a, M b, int *c) {
  modifies *a,b,*c;
  ensures  (*c)' = (*c)^ + 1 ∧ trashed(b)
           ∧ ¬trashed(*c)
           ∧ (if (*a)^ != (*c)^ then
               then ¬trashed(*a) ∧ (*a)' = (*c)^
               else trashed(*a));
}
```

(Notice the addition to the ensures clause of a predicate asserting that *c is not trashed.) In practice, very few functions trash the objects in their modified sets. For example, of the fifty-two functions given in LCL specifications in the Larch book, only two of the thirty-two expressions (that occur in the modifies clauses) are arguments to the **trashed** operator [8]. Thus, requiring an explicit statement of the fact that objects are preserved would (unnecessarily) lengthen specifications; function specifications that are less concise are more difficult to write, understand and maintain.

Fortunately there is a better solution. We suggest the introduction of a trashes clause which is syntactically like the modifies clause except for the leading **trashes** keyword. That is, the trashes clause is optional and when present, it may be followed by the **nothing** keyword, or by a list of lvalues (expressions denoting objects). A function may trash an object if and only if that object is referenced by an expression that occurs in the trashes clause[11]. Thus, the modifies clause recovers its intended role: it identifies which objects may have their values *modified*. The roles of the modifies and trashes clauses are *independent*; an expression may occur in both, in either or neither of the clauses. Under this scheme, the specification of **trashSome** would be identical to its original specification but with the addition of the clause **trashes** *a,b. Most function

[11] Actually, object dependencies must be taken into account for both the modifies and trashes clauses. Details are given in [2].

specifications will be written without a trashes clause, implying that no (client-visible) object may be trashed. For those few functions that do trash objects, these objects will be explicitly identified by listing them in the trashes clause.

6 Conclusion

The specialization of a specification language to a particular programming language is an important characteristic of module interface specification languages (MISL's). The only well-developed MISL's are the Larch interface languages and among these LCL would seem to be the most mature. We have argued that MISL's are an excellent way of introducing formal methods into industrial settings.

We have identified inadequacies and insufficiencies in the LCL language. In particular, by introducing the concept of object dependency we illustrate, by means of realistic examples, that there is a need for LCL language constructs that would allow specifiers to describe and reason about object dependencies. We argue that the meaning of a function specification is affected by implicit parameter constraints that have been poorly documented. These constraints are shown to be problematic—in particular, they are ambiguous and potentially overly constraining. We show that the current definition of the meaning of a function specification relative to trashed and non-trashed objects leads to a violation of the principle of referential transparency.

The version of LCL described in this paper differs from LCL 2.4, principally in that:

- new primitives have been added for describing object dependencies,
- the implicit constraints over pointer and array parameters have been dropped and new language primitives have been added that allow specifiers to assert whether or not an object is active or well-defined,
- a trashes clause has been added to function specification bodies.

These changes increase the expressiveness of LCL and allow us to overcome the identified shortcomings of LCL 2.4. In particular, we eliminate the instance of referential opacity. The shortcomings and solutions documented in this paper, as well as others that require a deeper understanding of the semantics of LCL, are described in detail in [2], which also includes a formal semantics for a core subset of LCL. Finally, we note that the identified shortcomings are not particular to LCL, they are shared by other module interface specification languages.

Acknowledgments

We thank Gary Leavens and David Evans for their comments on an earlier draft of this paper.

References

1. Jonathan Bowen and Mike Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.
2. Patrice Chalin. On the language design and semantic foundation of LCL, a Larch/C interface specification language. CU/DCS TR 95-12, Computer Science Department, Concordia University, December 1995. Ph.D. Thesis.
3. Jolly Chen. The Larch/Generic interface language. S. B. Thesis, Department of Electrical Engineering and Computer Science, MIT, 1989.
4. George W. Ernst, Raymond J. Hookway, and William F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Transactions on Software Engineering*, 20(4):288–307, April 1994.
5. David Evans. Using specifications to check source code. TR 628, MIT LCS, June 1994. S.M. Thesis.
6. David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Symposium on the Foundations of Software Engineering*, December 1994.
7. John V. Guttag and James J. Horning. LCL: A Larch interface language for C. Technical Report 74, DEC Systems Research Center, July 1991.
8. John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.
9. C.A.R. Hoare. An overview of some formal methods for program design. *IEEE Computer*, 20(9):85–91, September 1987.
10. Michal Iglewski, Jan Madey, David Lorge Parnas, and Philip C. Kelly. Documentation paradigms. CRL TR 270, McMaster University, July 1993.
11. ISO/IEC 9899 : 1990 (E). *Programming languages—C.*
12. Ann Jackson and Daniel Hoffman. Inspecting module interface specifications. *Software Testing, Verification and Reliability*, 4:101–117, 1994.
13. Cliff B. Jones. *Systematic Software Development using VDM.* Computer Science Series. Prentice Hall International, second edition, 1990.
14. Richard Allen Lerner. *Specifying Objects of Concurrent Systems.* PhD thesis, Carngie Mellon University, May 1991. TR CMU–CS–91–131.
15. Carroll Morgan. *Programming from Specifications.* Computer Science Series. Prentice Hall International, 1990.
16. David Lorge Parnas and Yabo Wang. The trace assertion method of module interface specification. TR 89-261, Queen's University at Kingston (Dept. of Computing and Information Science), 1989.
17. S. Prehn and W.J. Toetenel, editors. *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science.* VDM Europe, Springer-Verlag, 1991. Volume 1: Conference Contributions.
18. J.M. Spivey. *The Z Notation: A Reference Manual.* Computer Science Series. Prentice Hall International, second edition, 1992.
19. Yang Meng Tan. Formal specification techniques for promoting software modularity, enhancing documentation, and testing specifications. TR 619, MIT LCS, June 1994. Ph.D. Thesis.
20. Mark T. Vandevoorde. Exploiting specifications to improve program performance. TR 598, MIT LCS, February 1994. Ph.D. Thesis.
21. Jeannette M. Wing and Amy Moormann Zaremski. Unintrusive ways to integrate formal specifications in practice. In *[17]*, pages 545–569, 1991.