

# Reasoning about Memory Layouts

Holger Gast

Wilhelm-Schickard-Institut für Informatik  
University of Tübingen  
gast@informatik.uni-tuebingen.de

**Abstract.** Verification methods for memory-manipulating C programs need to address not only well-typed programs that respect invariants such as the split heap memory model, but also programs that access through pointers arbitrary memory objects such as local variables, single struct fields, or arrays slices. We present a logic for memory layouts that covers these applications and show how proof obligations arising during the verification can be discharged automatically using the layouts.

## 1 Introduction

Verification methods of programs that manipulate the heap necessarily formalize and reason about the memory layout: each access to the memory generates the proof obligation that the accessed region is allocated, and the influence of writes on the validity of assertions needs to be determined by considering the possible aliasing between pointers. The required reasoning has been automated successfully for Burstall’s split heap memory model, which in particular is expressive enough for object-oriented programming languages (e.g. [1, 2]). Single objects as well as sets of objects are supported by current reasoning technology (e.g. [3]).

Burstall’s memory model assumes that all pointers are object references and objects with different references do not overlap. It is therefore too imprecise for many C programs, and the employed reasoning techniques do not scale directly to more precise memory models [4]. Unfortunately, the excluded “low-level” usage is not confined to a few border cases, but is well within the range of idiomatic C code. A few examples from the current Linux kernel, which are deliberately taken from different modules, will illustrate the point.

Data structures throughout the kernel are, for instance, protected by mutexes. The following functions (from `mutex.c`) acquire and release mutexes.

```
void mutex_lock(struct mutex *lock);  
void mutex_unlock(struct mutex *lock);
```

The mutexes, i.e. memory objects of type `struct mutex`, are allocated in various ways. In `socket.c`, for instance, global mutexes protect the (global) `ioctl` settings. Calls like `mutex_lock(&br_ioctl_mutex)` are thus distributed throughout the module. But mutexes also protect `inodes` (defined in `fs.h`):

```
struct inode { ... struct mutex i_mutex; ... }      (1)
```

Locking such an `inode` involves passing a pointer to a struct member:

```
mutex_lock(&inode->i_mutex);
```

 (2)

Furthermore, it is common to pass pointers to local variables and also to fields within local variables (from `hrtimer.c`, where `struct hrtimer_sleeper t`):

```
hrtimer_init_on_stack(&t.timer, /*... */);
```

 (3)

Also elements from local arrays are passed by reference (from `compat.h`, where `struct timespec tv[2]`):

```
if (get_compat_timespec(&tv[0], &t[0])) { ... }
```

 (4)

Indeed, these examples do not represent particularly “low-level” kernel code. Similar idioms are presented in manuals and textbooks as the established best practice.

The common challenge in these examples is that the specifications of the called functions do not foresee these particular uses, but are formulated with respect to the passed pointers alone — they are small specifications [5]. To verify the calls, it is necessary to reason about the layout of the data structures and their components, and to derive frame axioms for the remaining data structures.

This paper’s contribution is a method for automatic reasoning about the above situations within Hoare logic. We provide a language for expressing layouts and a logic and proof method for refining and re-interpreting layouts. The approach is flexible in that it supports user-defined layout components and user-provided refinements and re-interpretations. This contribution is thus complementary to the work presented in [6], where unfoldings were left as future work. The treatment of layouts is also independent of the specific Hoare logic used; instead, it solves proof obligations which generally arise in Hoare logics.

The development presented in this paper is mechanized in Isabelle/HOL to ensure its soundness. We also use Isabelle/HOL as an example verification environment, and the presented proof strategies are implemented as ML tactics to establish their utility. However, a second perspective is possible: the theorems that are used for verification can be seen as a first-order axiomatization of the introduced layout constants and operators. In this perspective, HOL serves as a meta-logic in which these theorems are proven (see [7] for a similar discussion). We will emphasize this connection throughout the presentation.

*Organization of the Paper* Section 2 analyzes the proof obligations about memory layouts arising in Hoare logics and summarizes the main concepts from [6]. Section 3 gives a framework for reasoning about memory layouts that covers both refinements and re-interpretations. Section 4 shows that the framework can solve the introductory examples. Section 5 surveys related work. Section 6 concludes.

## 2 Memory-Related Proof Obligations in the Hoare Logic

This section summarizes the considered programming language and Hoare logic. It then examines the proof obligations resulting from the use of a low-level memory model and introduces our formalization of memory layouts. For brevity, the presentation elides the less important definitions from [6].

### 2.1 Language and Hoare Logic

The language that we consider is inspired by Norrish’s detailed analysis of C [8]. Its expressions include the usual primitive arithmetic operations, pointer dereferencing, and side-effecting operators, as well as pointer arithmetic and an address operator applicable to arbitrary l-values (i.e. memory objects). As statements, we support if, while, return, blocks with local variable declarations, and the execution of expressions. The syntax is the same as in C.

We use a standard big-step operational semantics. Compared with [8], we introduce mainly two simplifications with the purpose of focussing on memory-related aspects: first, expressions are executed left-to-right and side-effects are committed to memory immediately. Second, there is no distinction between allocated and initialized memory (cf. [8, Sec. 3.1.2] for both).

The memory model is captured by the following Isabelle/HOL type, where `addr` is a type isomorphic to 32-bit words [9]. (“ $\Rightarrow$ ” denotes total functions.)

```
record memory =  
  m-dom :: "addr set"  
  m-cnt  :: "addr  $\Rightarrow$  byte"  
  m-valid :: "bool"
```

A memory state’s *domain* and *content* together define a partial function from the allocated addresses to their content. The *history variable* `m-valid` designates whether any illegal accesses have occurred during the execution [10]. The operational semantics accesses memory only through the functions `fetch` and `store`, which transfer byte representations of values from and to memory. These functions set `m-valid` to `false` if unallocated addresses are manipulated.

```
fetch :: "addr  $\Rightarrow$  nat  $\Rightarrow$  memory  $\Rightarrow$  byte list  $\times$  memory"  
store :: "addr  $\Rightarrow$  byte list  $\Rightarrow$  memory  $\Rightarrow$  memory"
```

Execution is defined relative to a context, given by the following record type, which contains the definitions of struct types, functions, and local variables (“ $\rightarrow$ ” denotes partial functions; `ty` is the datatype representing the language types).

```
record ctx =  
  ctx-structs :: "string  $\rightarrow$  struct-def"  
  ctx-prog    :: "string  $\rightarrow$  func"  
  ctx-vars    :: "string  $\rightarrow$  addr  $\times$  ty"
```

Note that this memory model does not make a structural distinction between local variables and the heap. In particular, it is possible to apply the address operator and pointer arithmetic for accessing local variables.

We use a Hoare logic for fault-avoiding partial correctness. The rules are forward-style and generalize Floyd’s assignment axiom [6]. The treatment of

recursive functions and auxiliary variables are based on Schirmer’s presentation [11]. Side-effecting expressions are handled using Kowaltowski’s approach [12]. For the purposes of the present paper, only the memory-related proof obligations are important. They are analyzed subsequently.

## 2.2 Formalizing Layouts

Memory layouts are usually perceived as recursively nested objects (e.g. [10, 13]), which suggests a formalization by a grammar (or equivalently an algebraic data type). This approach has, however, the drawback that it fixes the set of possible memory layouts. The examples in Section 1, on the other hand, suggest that different views on a single memory state may be necessary. We therefore use a shallow embedding of memory layouts into HOL, i.e. we define HOL constants and functions that capture the memory region covered by a layout. The central notion is therefore that of a *cover*, which describes a region by comprehension:

cover = "addr set  $\Rightarrow$  bool"

All covers mentioned subsequently will be *well-formed* in the sense that they accept a single address set or none at all. It is then straightforward to define raw memory regions, and the regions occupied by some typed value, or a variable, and by arrays.<sup>1</sup>

block a n  $\equiv \lambda S. S = \{a .. < a \oplus n\} \wedge a \leq a \oplus n$   
typed-block  $\Gamma$  a t  $\equiv \lambda S. \text{block } a \text{ (of-nat (sz-of-ty } \Gamma \text{ t)) } S \wedge \text{is-small-type } \Gamma \text{ t}$   
var-block  $\Gamma$  v  $\equiv \text{typed-block } \Gamma \text{ (addr-of } \Gamma \text{ v) (type-of } \Gamma \text{ v)}$   
array  $\Gamma$  t p i j  $\equiv \lambda S. S = \{p \oplus [t]i .. < p \oplus [t]j\} \wedge 0 \leq_s i \wedge i \leq_s j \wedge$   
 $p \oplus [t]i \leq p \oplus [t]j \wedge \text{unat } j * (\text{sz-of-ty } \Gamma \text{ t}) \leq \text{unat max-word}$

The covers `block a n` and `array  $\Gamma$  t p i j` thus describe continuous regions of addresses. The side-conditions exclude overflows in the address arithmetic. The remaining two constants introduce typed views on blocks. Composite structures are expressed using the following disjointness combinator for covers:

$A \parallel B \equiv \lambda S. \exists S1 S2. A S1 \wedge B S2 \wedge S = S1 \cup S2 \wedge S1 \cap S2 = \{\}$

Subsequently, a *layout block* is a cover given by a defined constant, as opposed to being constructed by the disjointness combinator.

As an example, a variable `p` (of type `int*`), and the region it refers to would be formalized as follows (double quotes surround strings; `to_ptr` converts the byte representation of the pointer into an address; `rdv` reads the byte representation of the value stored in a variable):

var-block  $\Gamma$  "p"  $\parallel$  typed-block  $\Gamma$  (to\_ptr (rdv  $\Gamma$  "p")) int

In a first-order setting, the type `cover` would be taken as primitive. The introduced constants then become first-order functions and they are used in first-order axioms about layouts, as shown subsequently.

<sup>1</sup> `sz-of-ty`, `addr-of`, and `type-of` look up information on types and variables; `of-nat` and `unat` convert between `nat` and `word` [9]; `is-small-type` asserts that the size of a type can be represented in 32 bits; `max-word` is the the largest 32-bit word.  $\lambda$  denotes a function;  $\{a .. < b\}$  is the Isabelle/HOL notation for the interval  $[a, b)$ ; relations  $\leq_s$  and  $<_s$  are signed comparison on words [9],  $\oplus$  is raw address arithmetic, pointer arithmetic  $\oplus[\Gamma, t]$  uses a type and the definition context.

### 2.3 Normal Form of Assertions

The essence of lightweight separation [6] is that the user simply specifies the memory layout in addition to a first-order (or higher-order) assertion about the memory content. The memory layout is captured by covers, using the following *covered* predicate (read “M is covered by A”):

$$M \blacktriangleright A \equiv \text{m-valid } M \wedge A (\text{m-dom } M)$$

For an assertion P about the content, the normal form of assertions is therefore:

$$\lambda \Gamma M. \exists x_1 \dots x_n. M \blacktriangleright A \wedge P \Gamma M x_1 \dots x_n \quad (5)$$

The variables  $x_1 \dots x_n$  name intermediate results encountered during expression evaluation as usual in forward-style Hoare logics. In post-conditions of expressions, the result  $v$  would be an additional parameter [12]. In a first-order setting,  $\Gamma, M$ , and possibly  $v$  would be allowed to occur free in A and P.

### 2.4 Proof Obligations on Allocatedness

Since the Hoare logic is fault-avoiding, any memory access generates the proof obligation that the region is allocated. This condition is captured by the *allocated* predicate (read “A is allocated in M”, where A is a cover and M a memory state):

$$M \triangleright A \equiv \text{m-valid } M \wedge (\exists S. S \subseteq \text{m-dom } M \wedge A S)$$

In the rule for dereference expressions  $*e$ , for instance, let P be the post-condition of the evaluation of  $e$ . Following [12], it is a predicate on the current context  $\Gamma$ , the memory state M after the possibly side-effecting execution of  $e$ , and the computed result  $v$ . The necessary proof obligation is:

$$\forall \Gamma M v. P \Gamma M v \longrightarrow M \triangleright \text{typed-block } \Gamma (\text{to-ptr } v) t \quad (6)$$

By construction of the Hoare rules, P will be in normal form and the allocatedness can be determined from the layout given in P. The *subcover* relation captures just the necessary inclusion of the covered regions:

$$A \preceq B \equiv \forall S. B S \longrightarrow (\exists S'. S' \subseteq S \wedge A S')$$

The following theorem is then used to reduce (6) to a proof about layouts alone.

$$\frac{M \blacktriangleright A \quad B \preceq A}{M \triangleright B} \quad (7)$$

### 2.5 Side-conditions in Memory Layouts

Memory layouts usually include tacit assumptions about the involved addresses. The C standard prescribes, for instance, that an allocated block consists of a sequence of increasing addresses, which motivates the side-condition excluding overflows in the address arithmetic in the definition of `block` and `array` (Section 2.2). For the purposes of verification, these assumptions constitute invariants. Including them into the definitions of cover constants facilitates automatic

reasoning, since theorems about the constants have fewer premises. The definition of a cover constant  $c$  with parameters  $x_1 \dots x_n$  therefore usually has the following form (where the  $x_1 \dots x_n$ , but not the covered region  $S$ , occur in  $P$ ).

$$c x_1 \dots x_n \equiv \lambda S. S = \dots \wedge P x_1 \dots x_n$$

To express that the side-conditions hold, the covered region itself is immaterial. We say that a cover is *valid* if it covers some memory region; from the validity, it can be deduced that the side-conditions are satisfied.

$$\text{is-valid } A \equiv \exists S. A S$$

It is obviously possible to derive validity from a given memory layout by means of the subcover relation; furthermore, the subcover relation itself preserves validity.

$$\frac{M \triangleright A \quad B \preceq A}{\text{is-valid } B} \qquad \frac{\text{is-valid } A \quad B \preceq A}{\text{is-valid } B} \quad (8)$$

With the interpretation of side-conditions as invariants, an *is-valid* statement should be read as “the side-conditions hold”. In Section 3, this reading explains how side-conditions are maintained through unfoldings.

## 2.6 Side-effects, Disjointness, and Aliasing

Side-effects in the semantics are formalized in Hoare rules by syntactic manipulations of assertions. Suppose, for instance, that a command  $c$  performs some side-effect  $f$  on the memory, i.e. for the pre-state  $M$ , the post-state is  $f M$ . In a higher-order setting, the Hoare rule can be given directly (cf. [11, Figure 3.1]):

$$\vdash \{ \lambda \Gamma M. Q \Gamma (f M) \} c \{ Q \}$$

In a first-order verification environment, the  $\beta$ -reduction is replaced by a syntactic substitution, as in Hoare’s assignment axiom.

We prefer to use a forward-style Hoare logic in order to emulate the reasoning possible in separation logic [5, 7]. Here, Floyd’s assignment axiom can be generalized by introducing *inverse operators* [6]. Let  $F$  be a function such that  $F M (f M) = M$ , i.e.  $F$  undoes the effect of  $f$  by replacing a specific region by the content from its first argument. The forward-style Hoare rule for command  $c$  is then obtained by existentially quantifying over the previous memory state  $M'$ :

$$\vdash \{ P \} c \{ \lambda \Gamma M. \exists M'. P \Gamma (F M' M) \wedge Q \Gamma M \}$$

The assertion  $Q$  expresses the result of  $f$ , such as a particular region now containing a particular value. Again, in a first-order system, the verification condition generator would apply a syntactic substitution.

In both cases, therefore, the “current” memory state  $M$  in some assertion  $P$  is replaced by a modified state  $(g M)$ , where  $g$  is either the effect itself or its inverse operator. In both cases, the goal must be to remove the operator  $g$  in order to retrieve an assertion about the “current” state  $M$  itself.

To illustrate the point, suppose  $f$  is the operator  $\text{store } a \ v \ M$  which writes the byte-representation of value  $v$  at address  $a$  in  $M$ . Suppose then that precondition

P contains the assertion  $\text{to-int}(\text{rdv } \Gamma "x" M) > 0$ , which reads the content of variable "x" and interprets the byte-representation as an integer. The postcondition would be  $\text{to-int}(\text{rdv } \Gamma "x" (\text{STORE } a (\text{length } v) M' M)) > 0$  instead. Intuitively, this assertion reads: “before the side-effect, x contained a positive value”.

We thus wish to simplify terms of the form  $\text{mac}(\text{mop } M)$ , where  $\text{mac}$  is a *memory accessor* like  $\text{rdv } \Gamma x$  and  $\text{mop}$  is a *memory operator* like  $\text{STORE } a (\text{length } v) M'$ . We capture the behaviour of memory accessors and operators abstractly by the following constants (where the predicates  $\text{eqv-inside}$  and  $\text{eqv-outside}$  assert that their memory arguments are the same inside and outside, respectively, a region covered by A).

$$\begin{aligned} \text{accesses mac } M A &\equiv \forall M'. \text{eqv-inside } A M M' \longrightarrow \text{mac } M = \text{mac } M' \\ \text{modifies mop } M A &\equiv \text{eqv-outside } A M (\text{mop } M) \end{aligned}$$

Theorem (9) then allows the desired simplification to take place. Premises 1 and 2 are properties of the involved accessor and modifier constants. Premises 3 and 4 are solved automatically, since all covers we use are well-formed.

$$\frac{\begin{array}{l} \text{is-valid } B \longrightarrow \text{accesses mac } M B \\ \text{is-valid } A \longrightarrow \text{modifies mop } M A \\ \text{wf-cover } A \quad \text{wf-cover } B \\ M \blacktriangleright C \quad A \parallel B \preceq C \end{array}}{\text{mac}(\text{mop } M) = \text{mac } M} \quad (9)$$

In a first-order setting, the verification condition generator would use (9) to pre-generate rewrite rules by solving the premises 1–4 with provided theorems.

## 2.7 Function Calls

The specification of functions introduces the frame inference problem (e.g. [7]): it must be possible to infer from the specification which parts of the memory remain unmodified. As the examples in Section 1 show, no reasonable restriction can be placed on the memory objects that can be passed by reference. Our solution is to introduce a constant frame which asserts that a memory region given by a cover R has not been modified between the states M and M'.

$$\text{frame } R M M' \equiv \text{wf-cover } R \wedge \text{eqv-inside } R M M'$$

A function specification then consists of the pre- and post-conditions of the form

$$\begin{aligned} M \blacktriangleright A \parallel R \wedge \text{frame } R M_0 M \wedge P \\ M \blacktriangleright B \parallel R \wedge \text{frame } R M_0 M \wedge Q \end{aligned}$$

where A and B capture the memory parts directly manipulated by the function and R and M0 are auxiliary variables.

The pre-condition of the function call must then imply the function’s precondition. For a pre-condition  $\exists x_1 \dots x_n. M \blacktriangleright C \wedge P'$ , the proof obligation becomes:

$$\forall x_1 \dots x_n. M \blacktriangleright C \wedge P' \longrightarrow M \blacktriangleright A \parallel ?R \wedge \text{frame } ?R ?M_0 M \wedge P$$

Here the auxiliary variables have become unknowns ?R and ?M0 that can be instantiated (cf. [11, Sec. 3.1.1]). The reasoning task is expressed by theorem (10): we need to rewrite the cover C suitably, assuming that its invariants hold.

$$\frac{M \blacktriangleright C \quad \text{is-valid } C \longrightarrow C = A}{M \blacktriangleright A} \quad (10)$$

### 3 A Framework for Reasoning about Memory Layouts

Section 2 has identified the memory-related proof obligations arising in Hoare logics and has reduced them to subcover and equality relations between (first-order) covers. However, the challenges from Section 1 remain: in each of these examples, the manipulated memory parts are not directly given in the specified layout; instead, the layout must first be refined and re-interpreted. This section presents a matching framework for general *unfoldings* of memory layouts.

#### 3.1 Unfoldings of Layouts

In the following examples, we use the syntax supported by our Isabelle/HOL implementation: a variable block for  $x$  is written  $\langle\langle x \rangle\rangle$  and a typed block at  $p$  with type  $t$  is rendered as  $\langle\langle p:t \rangle\rangle$ ;  $\langle\langle t \rangle\rangle$  alone denotes type  $t$ . Context arguments,  $\text{rdv}$ ,  $\text{to-ptr}$ , etc. are inserted by translation functions [14, Sec. 8.6].

In the example (2), then, the local variable `inode` points to a record containing a mutex object, while the called function expects the mutex object alone. The required equality in (10) becomes:

$$\langle\langle \text{inode} \rangle\rangle \parallel \langle\langle \text{inode} : \text{struct inode} \rangle\rangle = \langle\langle \&\text{inode} \rightarrow \text{i\_mutex} : \text{struct mutex} \rangle\rangle \parallel ?R$$

The task is to unfold  $\langle\langle \text{inode} : \text{struct inode} \rangle\rangle$  into the constituent fields to expose the mutex. The examples (3) and (4) do, indeed, not introduce any new complications, because Hoare logic and memory layouts treat heap- and stack-allocated memory objects in the same way. The required equality for (3) is

$$\langle\langle t \rangle\rangle = \langle\langle \&t.\text{timer} : \text{struct hrtimer} \rangle\rangle \parallel ?R$$

In the final example (4), the only difference is that the variable `tv` is of type `struct timespec [2]`, i.e. contains an array rather than a struct:

$$\langle\langle \text{tv} \rangle\rangle = \langle\langle \&\text{tv}[0] : \text{struct timespec} \rangle\rangle \parallel ?R$$

The automated reasoning support must thus be able to re-write memory layouts on the fly. The unfolding rules used in our framework are of the form:

$$\frac{P_1 \dots P_n}{\text{is-valid } A \longrightarrow A = B_1 \parallel \dots \parallel B_m} \quad (11)$$

Whenever the premises  $P_1 \dots P_n$  hold, the layout  $A$  can be refined into layout  $B_1 \parallel \dots \parallel B_m$ . In the case  $m = 1$ , the unfolding is, in fact, a re-interpretation of layout block  $A$ . Note also how the side-conditions associated with  $A$  (Section 2.5) are available during the unfolding — the framework will apply the rule only in corresponding situations. The invariants associated with  $A$  therefore need not be repeated in the premises  $P_1 \dots P_n$ . If a theorem does not depend on the validity of  $A$ , it can omit the implication in the conclusion.

As a first example, variables can be re-interpreted as typed blocks by (12). When the program takes the address of a variable, the automated reasoning will apply the theorem correspondingly.

$$\frac{a = \text{addr-of } \Gamma v \quad t = \text{type-of } \Gamma v}{\text{var-block } \Gamma v = \text{typed-block } \Gamma a t} \quad (12)$$

By (13), unfolding rules can also be used to prove subcover relations as needed for allocatedness (7) and disjointness (9) proofs.

$$\frac{\text{is-valid } A \longrightarrow A = B}{B \preceq A} \quad (13)$$

By reflexivity of the equality and subcover relations and the following theorems, unfoldings can be applied anywhere within a nested layout:

$$\frac{\frac{\text{is-valid } A \longrightarrow A = A'; \quad \text{is-valid } B \longrightarrow B = B'}{\text{is-valid } (A \parallel B) \longrightarrow A \parallel B = A' \parallel B'}}{\frac{A \preceq A' \quad B \preceq B'}{A \parallel B \preceq A' \parallel B'}} \quad (14)$$

In principle, the examples from Section 1 can be solved using the above theorems and the unfolding rules for special data structures from Section 4. For restricted assertion languages, a brute-force unfolding with all possible rules is viable [7]. For first-order (or higher-order) assertions, we develop a two-step proof search. For a given proof obligation  $B \preceq A$  or  $\text{is-valid } A \longrightarrow A = B$ , it first *locates* all elementary blocks from  $B$  in  $A$  and then *unfolds*  $A$  to expose the blocks  $B$ , using the information gathered in the first step during the second.

### 3.2 Locating Memory Blocks

The proof obligations to be treated have the form  $B_1 \parallel \dots \parallel B_m \preceq A_1 \parallel \dots \parallel A_n$  or  $\text{is-valid}(A_1 \parallel \dots \parallel A_n) \longrightarrow A_1 \parallel \dots \parallel A_n = B_1 \parallel \dots \parallel B_m$ . For the proof search, it is useful to consider the blocks  $\{B_1 \dots B_m\}$  as a multiset by associativity and commutativity of disjointness. By abuse of notation, we will therefore write  $\{B_1 \dots B_m\} \preceq A$  for layout blocks  $B_1 \dots B_m$  and cover  $A$ . The location phase enriches this notation further with a justification of the subcover relation. In our Isabelle/HOL implementation of the proof search, this information is kept in ML data structures; for using a first-order prover, it can be encoded in terms.

The justification for a subcover relation  $B_k \preceq A$  (for  $k \in [1, m]$ ) consists in unfolding  $A$  in particular positions. Positions are denoted by *paths* in  $(L|R)^*$  (for “left” and “right”; the empty path is  $\varepsilon$ , concatenation is written  $p \cdot q$ ; the sub-layout of  $A$  at  $p$  is  $A \downarrow_p$ ). Since unfoldings can occur recursively, the enriched notation is:

$$\left\{ (B_k, p_{k0}, ((u_{k1}, p_{k1}) \dots (u_{km_k}, p_{km_k}))) \right\}_{k=1}^m \preceq A \quad (15)$$

The reading, to be defined formally in Section 3.3, is: for each  $k$ , the block  $B_k$  is found inside  $A$  by first following path  $p_{k0}$ , then applying unfolding rule  $u_{k1}$  (using (13)), then following  $p_{k1}$ , and proceeding in this manner until all unfoldings and paths have been exhausted. Note that for each application of an unfolding rule  $u_{kj}$ , its premises need to be proven (see (11)).

We now show that justifications can be computed efficiently. The central idea is to prepare in advance a set  $S$  for subcover justifications of all possible combinations of a given set of unfolding rules. The process starts with the trivial

justification  $\{(A, \varepsilon, \varepsilon)\} \preceq A$ , which captures the reflexivity of the subcover relation. Then, for any  $(\{(C, \varepsilon, J)\} \preceq D) \in S$  and unfolding step  $u$  of the form (11), where  $\sigma C = \sigma A$  for some  $\sigma$ , i.e.  $C$  unifies with  $A$ , we insert for  $k = 1 \dots m$  into  $S$  a new justification

$$\{(\sigma \mathbf{B}_k, \varepsilon, ((\sigma u, p_k), \sigma J))\} \preceq \sigma D$$

The application of substitution  $\sigma$  here is defined by its application to all occurring terms and theorems. The path  $p_k$  is the path to  $\mathbf{B}_k$  on the right-hand side in (11). Note that these justifications are sound by lemmata (16).

$$\mathbf{C} \preceq \mathbf{C} \parallel \mathbf{D} \quad \mathbf{D} \preceq \mathbf{C} \parallel \mathbf{D} \quad (16)$$

If this saturation process terminates with a set  $S$ , then by construction  $B \preceq A$  can be proven by the given unfolding rules iff there is some path  $p_0$ , substitution  $\sigma$ , and justification  $(\{(B', \varepsilon, J)\} \preceq A') \in S$  such that  $B = \sigma B'$  and  $A \downarrow_{p_0} = \sigma A'$ . The resulting justification is then  $\{(B, p_0, \sigma J)\} \preceq A$ .

To solve the proof obligations given at the beginning of this section efficiently, just apply this step to all pairs  $\mathbf{A}_i$  and  $\mathbf{B}_j$ . A term index is used to identify possible justifications. The desired result (15) is found.

*Remark 1.* The generation phase might fail to terminate. This is, in particular, the case for recursive, linked data structures such as lists or trees, whose structural unfoldings can be applied several times in a row. The termination argument for the only approach handling such unfoldings relies on a restricted form of assertions that is not sufficient for full functional verification [7]. We therefore leave the question as future work.

*Remark 2.* If the unfoldings are not deterministic, a block  $\mathbf{B}_j$  might be located in  $\mathbf{A}_i$  by several justifications. (However, by definition of disjointness, it cannot be located in a different  $\mathbf{A}_{i'}$ .) In the subsequent presentation, we assume that a single justification has been computed; if ambiguities arise, they are resolved by iteration through the possible solutions.

### 3.3 Computing Unfoldings

The result of the location phase is of the form  $\{(\mathbf{B}_k, p_{0k}, J_k)\}_{k=1}^m \preceq \mathbf{A}$ : it captures precisely where the  $\mathbf{B}_1 \dots \mathbf{B}_m$  are located inside layout  $\mathbf{A}$ . To compute the actual unfolding equality, it is sufficient to follow the justifications  $\{(p_{0k}, J_k)\}_{k=1}^m$  in a recursive process. We give the process in the form of inference rules for a judgement

$$\{(\mathbf{B}_k, p_{0k}, J_k) \mid P\ k\} \rightsquigarrow \text{is-valid } \mathbf{A} \longrightarrow \mathbf{A} = \mathbf{B}$$

where  $P$  selects a subset of  $[1, m)$  and the result  $\text{is-valid } \mathbf{A} \longrightarrow \mathbf{A} = \mathbf{B}$  is a theorem. The recursion base consists in a single  $\mathbf{B}_k$  being actually found:

$$\{(\mathbf{B}_k, \varepsilon, \varepsilon)\} \rightsquigarrow \text{is-valid } \mathbf{B}_k \longrightarrow \mathbf{B}_k = \mathbf{B}_k \quad (17)$$

The first recursion step computes the unfoldings of two disjoint sub-multisets of the  $\mathbf{B}_k$  using theorem (14).

$$\frac{\begin{array}{l} \{(\mathbf{B}_k, p'_{0k}, J_k) | P k \wedge p_{0k} = L \cdot p'_{0k}\} \rightsquigarrow \text{is-valid } A \longrightarrow A = B \\ \{(\mathbf{B}_k, p'_{0k}, J_k) | P k \wedge p_{0k} = R \cdot p'_{0k}\} \rightsquigarrow \text{is-valid } A' \longrightarrow A' = B' \\ \forall k. P k \longrightarrow p_{0k} \neq \varepsilon \end{array}}{\{(\mathbf{B}_k, p_{0k}, J_k) | P k\} \rightsquigarrow \text{is-valid } (A \parallel A') \longrightarrow A \parallel A' = B \parallel B'} \quad (18)$$

Finally, if the paths in the justification have been exhausted, an unfolding takes place. We denote by  $u(A)$  the application of the unfolding rule of the form (11) to a term  $A$ , which yields the unfolded layout. At this point the premises of step  $u$  need to be proven.<sup>2</sup> Note how the local paths  $p_k$  become the new paths in the justification.

$$\frac{\{(\mathbf{B}_k, p_k, J_k) | P k\} \rightsquigarrow \text{is-valid } u(A) \longrightarrow u(A) = B}{\{(\mathbf{B}_k, \varepsilon, (u, p_k) \cdot J_k) | P k\} \rightsquigarrow \text{is-valid } A \longrightarrow A = B} \quad (19)$$

*Remark 3.* Since (18) and (19) apply only to judgments of specific forms, the completeness of this procedure must be discussed. Consider therefore a justification  $\{(\mathbf{B}_k, p_k, J_k) | P\} \preceq A$  where  $A$  is a layout block and the  $\mathbf{B}_k$  can be proven disjoint by the given unfolding rules. If the multiset is a singleton, rule (17) applies. Next, since the blocks  $\mathbf{B}_k$  can be proven disjoint, either all  $p_k$  must be non-empty, in which case (18) applies, or they are all empty. (Otherwise, there would be some  $k$  and  $k'$  with  $\mathbf{B}_{k'} \preceq \mathbf{B}_k$ , contradicting disjointness.) In this latter case, (19) applies with some common unfolding rule  $u$ , since we can assume that the justifications have been selected correspondingly by Remark 2.

The process of unfolding a memory layout is thus made deterministic by employing the information gathered in the location phase.

### 3.4 Unfolding On-Demand

The above presentation of the proof search assumes that unfolding rules have the static form (11). This is, however, too restrictive in general: locating a set of array elements and array slices in a given array would require “guessing” a suitable split of the index range in advance (see Section 4.2). The problem is solved by lazy computation of unfoldings. The unfolding rules are used only in two places: to generate the subcover theorems for the location phase by (13) and to compute the unfolding in (19). The first use can be removed if the subcover theorems are given directly. When the unfolding rule is required in (19), more information is available in the form of the proven premises of (11). We therefore apply a standard strategy (e.g. [14, Sec. 10.2.5]): when (19) is applied, an ML function is called with the current judgment. The function returns both the unfolding rule  $u$  and the paths  $p_k$  for each of the blocks in the left-hand side. Unfoldings in this way are computed lazily, on-demand.

<sup>2</sup> Note that the same premises have been proven during the location phase. In the ML implementation, we keep the proven premises as Isabelle theorems.

## 4 Applications

The framework from Section 3 supports automatic reasoning about a wide range of unfoldings of memory layouts. In this section, we apply it to examples derived from Section 1 by giving specific unfolding rules. For brevity, we can only show prototypical examples that focus on the main considerations of the paper.

### 4.1 Struct Types

The layout of a struct type is given with its type definition. It consists of the struct's fields, possibly separated by padding to ensure alignment of the fields' data. Since it can be added straightforwardly, we neglect padding for brevity. Using auxiliary constants `field-off` and `field-ty`, which determine the offset and type of a field in a given struct type, we can define a new cover for a single field:

$$\text{field-block } \Gamma \text{ p t f} \equiv \text{typed-block } \Gamma (\text{p} \oplus (\text{of-nat } (\text{field-off } \Gamma \text{ t f}))) (\text{field-ty } \Gamma \text{ t f})$$

For a list of fields, their joined layout is then given by a recursive function:

$$\begin{aligned} \text{fields-cover } \Gamma \text{ t a } [] &= \text{Empty} \\ \text{fields-cover } \Gamma \text{ t a } ((\text{f,ty}) \# \text{fs}) &= \text{field-block } \Gamma \text{ a t f} \parallel \text{fields-cover } \Gamma \text{ t a fs} \end{aligned}$$

With these preliminary definitions, we can prove a general unfolding theorem for struct types. Its premise captures that type `t` is a defined struct in the context  $\Gamma$ .

$$\frac{\text{wf-struct } \Gamma \text{ t}}{\text{typed-block } \Gamma \text{ a t} = \text{fields-cover } \Gamma \text{ t a } (\text{struct-fields } \Gamma \text{ t})} \quad (20)$$

For a specific struct type such as `struct point { int x; int y; }`, the special rule (21) can be generated from (20) directly. Here `point-known`  $\Gamma$  captures, again, that the definition of `struct point` is present in context  $\Gamma$ .

$$\frac{\text{point-known } \Gamma}{\text{typed-block } \Gamma \text{ a } (\text{struct point}) = \begin{array}{l} \text{field-block } \Gamma \text{ a } (\text{struct point}) \text{"x"} \parallel \\ \text{field-block } \Gamma \text{ a } (\text{struct point}) \text{"y"} \end{array}} \quad (21)$$

The re-interpretation of a local variable of struct type as a memory object is given by the unfolding step (12). Together with (21), the challenges of the introductory examples can be solved. Suppose, for instance, a function `void set(int *p, int i)` sets `*p` to `i`. Its precondition requires a layout  $M \blacktriangleright \langle\langle p : \text{int} \rangle\rangle \parallel R$ , the postcondition asserts that `*p = i` and `R` is framed, i.e. not modified. We can then verify the following triple automatically (where `struct point s`; and `int i`; are local variables;  $\langle\langle s.x \rangle\rangle$  is expanded to reading field `x` from struct variable `s`).

$$\begin{aligned} &\vdash \{ M \blacktriangleright \langle\langle s \rangle\rangle \parallel \langle\langle i \rangle\rangle \wedge \text{point-known } \Gamma \wedge P \langle\langle s.x \rangle\rangle i \} \\ &\quad \text{set}(\&s.y, 1); \\ &\{ M \blacktriangleright \langle\langle s \rangle\rangle \parallel \langle\langle i \rangle\rangle \wedge \text{point-known } \Gamma \wedge P \langle\langle s.x \rangle\rangle i \wedge \langle\langle s.y \rangle\rangle = 1 \} \end{aligned}$$

For the function call, the given layout is unfolded to reveal the field-block for `s.y` and to show that the remainder `R` of the memory consists of the field-block for `s.x` and the local variable `i`. Note that the higher-order predicate `P` represents an arbitrary further assertion about `s.x` and `i`, independently of its actual structure.

This example covers usage (3) directly. It also shows the simpler usage (2) to be supported, where the variable-block re-interpretation (12) can be omitted.

## 4.2 Arrays

The unfolding of arrays poses the problem that the necessary refinement of the layout cannot be determined in advance, because it depends on the actual slices and elements that need to be revealed. Unfoldings are therefore computed lazily (Section 3.4) and explicit subcover theorems are used in the location phase:<sup>3</sup> (22) and (23) allow the prover to locate slices and elements, and (27) re-interprets an array element as a typed block, thus enabling access by pointer arithmetic.

$$\frac{i \leq_s i' \quad i' <_s j' \quad j' \leq_s j}{\text{array } \Gamma \text{ t a } i' j' \preceq \text{array } \Gamma \text{ t a } i j} \quad (22)$$

$$\frac{i \leq_s j \quad j <_s k}{\text{array-elem } \Gamma \text{ t a } j \preceq \text{array } \Gamma \text{ t a } i k} \quad (23)$$

$$\frac{p = a \oplus [i, t] j}{\text{array-elem } \Gamma \text{ t a } j = \text{typed-block } \Gamma \text{ p t}} \quad (24)$$

With these rules, the location phase can associate layout blocks with a given array, proving the premises in the process. When the actual unfolding is required, an ML function is called to compute an unfolding rule tailored to the situation. Towards that end, it determines the relative order of the indices from the proven premises and uses the theorems (25) and (26).

$$\frac{i \leq_s j \quad j \leq_s k}{\text{array } \Gamma \text{ t a } i k = \text{array } \Gamma \text{ t a } i j \parallel \text{array } \Gamma \text{ t a } j k} \quad (25)$$

$$\text{is-valid}(\text{array } \Gamma \text{ t a } j (j+1)) \longrightarrow \text{array } \Gamma \text{ t a } j (j+1) = \text{array-elem } \Gamma \text{ t a } j \quad (26)$$

Theorem (26) is of particular interest, because it crucially uses the invariants associated with the single-element array. An array element is simply a typed-block (see (24)) which does not contain all invariants of arrays (Section 2.2).

Arrays in local variables, which have type  $\text{Array t n}$  with constant size  $n$  and element type  $t$ , are found by the following rule (27), which is applied after (12).

$$\text{is-valid}(\text{typed-block } \Gamma \text{ a } (\text{Array t n})) \longrightarrow \text{typed-block } \Gamma \text{ a } (\text{Array t n}) = \text{array } \Gamma \text{ t a } 0 n \quad (27)$$

Again, this unfolding relies on the validity of the typed block to establish the side-conditions of the `array` from the well-formedness of the `Array` type.

With these unfoldings, the last introductory challenge (4) can be resolved. Here, a local variable `int a[16]`; is allocated and some assertion about its first  $i$  elements is given. It can be proven automatically that setting `a[i]` does not influence that assertion, as stated in the following triple.

$$\begin{aligned} &\vdash \{ M \blacktriangleright \langle i \rangle \parallel \langle a \rangle \wedge 0 \leq_s i \wedge i <_s 16 \wedge (\forall k. 0 \leq_s k \wedge k <_s i \longrightarrow \langle a[k] \rangle = 0) \} \\ &\quad a[i] = 1; \\ &\{ M \blacktriangleright \langle i \rangle \parallel \langle a \rangle \wedge 0 \leq_s i \wedge i <_s 16 \wedge (\forall k. 0 \leq_s k \wedge k <_s i \longrightarrow \langle a[k] \rangle = 0) \} \end{aligned}$$

This example shows also that our method goes beyond automated fragments of separation logic [7, 15] in handling local assumptions on quantified variables.

<sup>3</sup> The implementation never applies the same rule twice in a row, which prevents non-termination with (22).

A final example demonstrates the flexibility of the presented framework. Suppose the function `void memcpy(char *src, char *dst, int n)` copies memory byte-wise. Its precondition demands that the source and destination arrays are allocated:

$$M \triangleright \text{array } \Gamma \langle\langle \text{char} \rangle\rangle \text{ src } 0 \ n \ \parallel \ \text{array } \Gamma \langle\langle \text{char} \rangle\rangle \text{ dst } 0 \ n \ \parallel \ R$$

Its post-condition asserts the same layout, that the elements of `src` have been copied to `dst`, and that only the `dst` array has been modified, such that any assertions about `src` and `R` continue to hold.

$$M \triangleright \text{array } \Gamma \langle\langle \text{char} \rangle\rangle \text{ src } 0 \ n \ \parallel \ \text{array } \Gamma \langle\langle \text{char} \rangle\rangle \text{ dst } 0 \ n \ \parallel \ R \wedge \\ \text{array-elems } \Gamma \langle\langle \text{char} \rangle\rangle \text{ src } 0 \ n \ M = \text{array-elems } \Gamma \langle\langle \text{char} \rangle\rangle \text{ dst } 0 \ n \ M \wedge \\ \text{frame } (R \ \parallel \ \text{array } \Gamma \langle\langle \text{char} \rangle\rangle \text{ src } 0 \ n) \ M 0 \ M$$

Then, we can implement a low-level string concatenation. We prove the following triple in a variable context `int m; int n; char *a; char *b; char *dst;`.

$$\vdash \{ M \triangleright \langle\langle a \rangle\rangle \ \parallel \ \langle\langle n \rangle\rangle \ \parallel \ \langle\langle b \rangle\rangle \ \parallel \ \langle\langle m \rangle\rangle \ \parallel \ \langle\langle \text{dst} \rangle\rangle \ \parallel \\ \text{array } \Gamma \langle\langle \text{char} \rangle\rangle \text{ a } 0 \ n \ \parallel \ \text{array } \Gamma \langle\langle \text{char} \rangle\rangle \text{ b } 0 \ m \ \parallel \ \text{array } \Gamma \langle\langle \text{char} \rangle\rangle \text{ dst } 0 \ (n + m) \wedge \\ \text{array-elems } \Gamma \langle\langle \text{char} \rangle\rangle \text{ a } 0 \ n \ M = A \wedge \text{array-elems } \Gamma \langle\langle \text{char} \rangle\rangle \text{ b } 0 \ m \ M = B \} \\ \text{memcpy}(a, \text{dst}, n); \\ \text{memcpy}(b, \text{dst} + n, m); \\ \{ \text{array-elems } \Gamma \langle\langle \text{char} \rangle\rangle \text{ dst } 0 \ (n + m) \ M = (A @ B) \}$$

In the second call, the precondition of `memcpy` mentions an array starting at index 0, while the actual argument is a slice starting at `n`. The re-interpretation (28) shifts a given array slice to begin at index 0.

$$\frac{b = a \oplus [l, t] \ i \quad m = n - i}{\text{is-valid } (\text{array } \Gamma \text{ t a i n}) \longrightarrow \text{array } \Gamma \text{ t a i n} = \text{array } \Gamma \text{ t b 0 m}} \quad (28)$$

## 5 Related Work

The problem of proving the disjointness of memory regions has recently attracted much attention in connection with the verification of object-oriented programs. Kassios [16] proposes to express the memory region occupied by an object's representation as an additional specification variable. The disjointness of the regions, hence the independence of assertions from particular memory operations, can then be asserted without breaking encapsulation. The approach has been implemented by several authors [3, 1]. The work addresses Burstall's memory model and uses direct pointer comparisons throughout. It does therefore not scale directly to more low-level memory models. Greve [17] proposes to express the memory region occupied by data structures by functions with the intention of proving disjointness of modified memory regions. Neither of these approaches addresses the problems of re-interpretation and structural refinement of memory layouts beyond the field-level access encompassed by Burstall's model.

Automatic unfoldings of memory layouts are supported by the Smallfoot [7] tool. It is based on a restricted form of separation logic that is suitable for expressing shape invariants of data structures. Recently, Tuerk [15] has shown that some assertions of the content of data structures can be handled in parallel with their structure. The unfolding mechanism in both works relies on

the fact that all assertions give rise to finitely many unfoldings, such that an undirected search is possible. Tuerk uses [18] to cover call-by-reference with entire local variables, but not with more general memory objects.

Tuch [10] applies separation logic to structured data types. In particular, he develops a theory for struct types in C. The automatic reasoning provided is limited: a specialized tactic allows the user to fold and unfold struct definitions as needed. Neither arrays nor references to local variables are supported. Cohen et al. [13] establish a typed memory model over untyped C memory states by expressing the additional disjointness invariants using a ghost variable. Special statements `split` and `join` in the language serve to manipulate the layout. The approach handles structs and is extensible to bit-fields and arrays. The frame inference problem for functions is not discussed.

## 6 Conclusion

We have presented a method for automatic reasoning about memory layouts in a flexible and extensible manner. A small language of layouts allows memory-related proof obligations arising in Hoare logics to be formulated succinctly. The actual elements of layouts are not fixed, but can be defined as needed: local variables, blocks accessed by pointers, structs, and arrays are readily formalized. Our reasoning framework supports a general notion of unfoldings of memory elements. Unfoldings comprise both structural refinements and re-interpretations of layout blocks. Structs and arrays can thus be split automatically into their constituent elements as needed, and local variables can be interpreted as memory objects, which allows them to be accessed by pointers in arbitrary ways. Introducing a new unfolding generally requires nothing more than proving an unfolding equality theorem about a cover constant. Using the flexibility, it is possible to verify idiomatic usages of C that are not currently covered by other verification methodologies.

The presented method is suitable for reasoning about a low-level, byte-addressed memory model. The key insight is to encode invariants about memory layouts into the definition of layout constants, and to propagate the invariants through unfoldings. As an illustration, the theory reduces reasoning about arrays to proving inequalities between indices without further side-conditions; overflows in the address arithmetic are excluded by the invariants associated with arrays.

Three future directions of the work appear promising. The first one is to apply standard first-order reasoners rather than specialized ML tactics. Even though the theory is developed in Isabelle/HOL to ensure soundness, first-order theorems are sufficient for the actual verification. A second direction is the verification of low-level programs using pointer-casts. The re-interpretations necessary when using casts can be expressed as unfolding theorems handled by the framework. Finally, the developed logic of memory layouts is largely independent of the employed Hoare logic. It would therefore be interesting to integrate the reasoning with an existing verification condition generator.

## References

1. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An automatic verifier for Java-like programs based on dynamic frames. In Fiadeiro, J.L., Inverardi, P., eds.: FASE. Volume 4961 of Lecture Notes in Computer Science., Springer (2008) 261–275
2. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of LNCS., Springer (2007) 173–177
3. Banerjee, A., Barnett, M., Naumann, D.A.: Boogie meets regions: A verification experience report. In Shankar, N., Woodcock, J., eds.: VSTTE'08. Volume 5295 of LNCS., Springer (2008) 177–191
4. Rakamarić, Z., Hu, A.J.: A scalable memory model for low-level code. In: 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009). (2009)
5. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Proceedings of the 15th International Workshop on Computer Science Logic. Number 2142 in LNCS, Springer (2001) 1–19
6. Gast, H.: Lightweight separation. In Ait Mohamed, O., Muñoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics 21st International Conference, TPHOLs 2008. Volume 5170 of LNCS., Springer (2008)
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO. Volume 4111 of LNCS., Springer (2005) 115–137
8. Norrish, M.: C formalised in HOL. PhD thesis, University of Cambridge (1998) Technical Report UCAM-CL-TR-453.
9. Dawson, J.E.: Isabelle theories for machine words. In: Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07). ENTCS (2007)
10. Tuch, H.: Structured types and separation logic. In: 3rd International Workshop on Systems Software Verification (SSV). (2008)
11. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2005)
12. Kowaltowski, T.: Axiomatic approach to side effects and general jumps. *Acta Informatica* **7** (1977) 357–360
13. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C. In: 4th International Workshop on Systems Software Verification (SSV). ENTCS. (2009)
14. Paulson, L.C.: Isabelle – A Generic Theorem Prover. Number 828 in LNCS. Springer, Berlin Heidelberg (1994)
15. Tuerk, T.: A formalisation of Smallfoot in HOL. In: Theorem Proving in Higher-Order Logics (TPHOLs 2009). (2009) to appear.
16. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of LNCS., Springer (2006) 268–283
17. Greve, D.: Scalable normalization for heap manipulating functions. In: International Workshop on the ACL2 Theorem Prover and its Applications. (2007)
18. Parkinson, M., Bornat, R., Calcagno, C.: Variables as resource in Hoare logics. In: LICS '06: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, IEEE Computer Society (2006) 137–146