Constraint-Based Contract Inference for Deductive Verification

Anoud Alshnakat[1], Dilian Gurov[1], Christian Lidström[1], and Philipp Rümmer[2(✉)]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
{anoud,dilian,clid}@kth.se
[2] Uppsala University, Uppsala, Sweden
philipp.ruemmer@it.uu.se

Abstract. Assertion-based software model checking refers to techniques that take a program annotated with logical assertions and statically verify that the assertions hold whenever program execution is at the corresponding control point. While the associated annotation overhead is relatively low, these techniques are typically monolithic in that they explore the state space of the whole program at once, and may therefore scale poorly to large programs. Deductive software verification, on the other hand, refers to techniques that prove the correctness of a piece of software against a detailed specification of what it is supposed to accomplish or compute. The associated verification techniques are modular and scale well to large code bases, but incur an annotation overhead that is often very high, which is a real obstacle for deductive verification to be adopted in industry on a wider scale. In this paper we explore synergies between the two mentioned paradigms, and in particular, investigate how interpolation-based Horn solvers used for software model checking can be instrumented to infer missing procedure contracts for use in deductive verification, thus aiding the programmer in the code annotation process. We summarise the main developments in the area of automated contract inference, and present our own experiments with contract inference for C programs, based on solving Horn clauses. To drive the inference process, we put program assertions in the main function, and adapt our TriCera tool, a model checker based on the Horn solver Eldarica, to infer candidate contracts for all other functions. The contracts are output in the ANSI C Specification Language (ACSL) format, and are then validated with the Frama-C deductive verification tool for C programs.

1 Introduction

Static approaches in program verification, including deductive verification [1,9] and model checking [8], offer an unparalleled level of confidence that software is indeed correct, and are receiving increasing attention in industrial applications. Static verification of a program can proceed in different ways: *monolithic* approaches receive the program and some form of specification as input, and attempt to construct a mathematical argument that the program, as a whole,

W. Ahrendt et al. (Eds.): Deductive Software Verification, LNCS 12345, pp. 149–176, 2020.
https://doi.org/10.1007/978-3-030-64354-6_6

satisfies the specification; *modular* approaches successively subdivide the verification problem into smaller and smaller parts, and thus construct a hierarchical correctness argument. In the verification literature, many examples in this spectrum from monolithic to modular methods can be identified; for instance, the state space of a concurrent program can be explored systematically by enumerating possible interleavings, a monolithic approach applied in classical model checking [21], but it is also possible to analyse the threads of a program one by one with the help of invariants, relies, or guarantees [22,31]. Monolithic and modular methods have complementary properties; monolithic methods tend to be easier to automate, while modular methods tend to scale to larger programs or systems.

In this paper, we explore synergies between monolithic and modular methods for the verification of programs with procedures. As monolithic methods, we consider model checkers built using the concept of *constrained Horn clauses* [6, 16,20]: such tools are able to verify, among others, programs with procedures and recursion fully automatically, and can in case of success output program artefacts including loop invariants and contracts [26]. On the modular side, we target deductive Hoare logic-based verification tools, which besides the program also needs detailed intermediate program annotations as input. Such specifications are written in a richer logical language, and are in principle to be supplied by the designer of the software, since they should express his or her intention. For procedural programming languages, they are given by means of procedure contracts that capture what each procedure is obliged to achieve when called, and under what assumptions on the caller.

We argue that these two families of tools complement each other extremely well: on the one hand, model checkers can automatically compute the program annotations required by deductive verification tools, and thus be used as invariant and contract inference tools; on the other hand, deductive verification tools can act as proof checkers that independently validate the computation of a model checker. Several hybrid combination approaches are possible as well: given a program partially annotated with invariants and contracts, a model checker could add the missing annotations, or show that the existing annotations are inconsistent and need corrections. For a program that is too large to be handled by a software model checker, manually provided contracts can be used to split the program into multiple parts of manageable size. For procedures that are invoked from multiple programs, contracts inferred in the context of one program can be reused for deductive verification of another program. The long-term vision of the presented line of research is the development of a *program annotation assistant* that applies fully automatic methods, including model checking, to infer, augment, or repair the annotations needed in deductive verification.

The main contributions of the paper are (i) a brief survey of the main directions in automatic contract inference (Sect. 3); (ii) the definition of the required program encoding and annotation translation to combine the Horn clause-based software model checker TriCera with the deductive verification system Frama-C [9] (Sects. 4 and 2); and (iii) an experimental evaluation of the performance of this tool combination on benchmarks taken from the SV-COMP [5] (Sect. 6).

```
int nondet();
/*@ contract @*/
int mc91(int n) {
  if (n > 100) {
    return n − 10;
  } else {
    return mc91(mc91(n + 11));
  }
}

int main() {
  int x = nondet();
  int res = mc91(x);
  assert(x!=8 || res==91);
  assert((x<=102) || res==x−10);
}
```

Listing 1.1. TriCera input for the McCarthy 91 function

```
/*@
//Function: mc91
    requires \true;
    assigns \nothing;
    ensures (n <= 100 ==>
               \result == 91)
    ensures (n > 100 ==>
               \result == n − 10);
*/
int mc91(int n) {
  if (n > 100) {
    return n − 10;
  } else {
    return mc91(mc91(n + 11));
  }
}
```

Listing 1.2. An example of ACSL contracts for Frama-C

Motivating Example. We first illustrate the relationship between software model checking and deductive verification, using a C-version of the well-known McCarthy 91 function as a motivating example:

$$\text{mc91}(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ \text{mc91}(\text{mc91}(n + 11)) & \text{if } n \leq 100. \end{cases}$$

Today's *software model checkers* focus mainly on the verification of safety (or reachability) properties embedded in program code in the form of assertions. Model checkers aim at fully automatic verification of programs, and therefore try to prove the absence of assertion violations without requiring any further code annotations. In the implementation of the McCarthy 91 function shown in Listing 1.1, two assertions in the main function capture the post-conditions to be verified: for the input 8, the result of the McCarthy 91 is conjectured to be 91, and for any input greater than 102 the result will be the input minus 10. The main function serves as a *harness* for the verification of the function of interest. The input in Listing 1.1 can be verified automatically by state-of-the-art model checkers, for instance by the TriCera model checker considered in this paper. TriCera can handle functions either by inlining them, or by inferring function contracts consisting of pre- and post-conditions. In our example, a contract is needed, since the McCarthy 91 function is recursive; the use of a contract is enabled by the comment `/*@ contract @*/`. The translation of the computed contract to the specification language ACSL is discussed in Sect. 5, and the contract is shown in Listing 5.2.

Deductive verification systems such as Frama-C [9] and KeY [1], in contrast, rely to a larger degree on user-provided annotations. In the case of the McCarthy 91 function, verification cannot proceed before a suitable function contract is supplied. Annotations and specifications in Frama-C are written in the behavioural specification language ACSL, which contains constructs written in special C comments. The function contract annotation begins with `/*@` and ends with `*/`, as shown in Listing 1.2. The keyword `ensures` is used to specify post-conditions. More details on Frama-C and ACSL will be given in Sect. 4.3. The contract given in Listing 1.2 is manually written, and gives a complete specification of the function. While most deductive verification tools are not able to compute contracts, they can usually check the correctness of contracts fully automatically.

ACSL annotations required by Frama-C, or similar tools, are detailed, and thus laborious to write. As the study [24] argues, an effective combination of contracting and function inlining is indispensable for the scalability of deductive verification. Going beyond this, automating the process of inferring function contracts is clearly advantageous for the software development process: earlier studies have shown that automation is a key enabler for the wider use of formal methods in industrial settings, and that automated annotation is of particular importance [28]. In this paper, we propose to leverage the ability of software model checkers to automatically compute contracts, and this way support the deductive verification process. That is, given an input specification and C code as in Listing 1.1, we want to generate a contract similar to Listing 1.2 for use in Frama-C. This paper provides the theoretic background for such a combination, and presents first practical steps.

2 The Notion of a Contract

In this section we recall the notion of a (software) contract, and define the operators of contract refinement and composition that provide the theoretical foundation for working with contracts.

2.1 Contract Basics

For reasons of presentation, all considerations are done in the context of a simple while-language with functions, with integers as the only data-type. We further assume that function parameters are passed by value and are read-only, and that programs do not contain global variables (or other kinds of global data, like heap). All results generalise to more realistic settings, like to the specification language ACSL used by Frama-C.

Definition 1 (Contract [26]). *Suppose f is a function with formal parameters $\langle a_1, \ldots, a_n \rangle$ and a formal result variable r. A* contract *for f is a pair $(Pre_f, Post_f)$ consisting of a pre-condition Pre_f over the arguments $a_1, \ldots, a_n$, and a post-condition $Post_f$ over the arguments $a_1, \ldots, a_n$ and the result r.*

Pre- and post-conditions are commonly represented as formulas in first-order logic, modulo a suitable set of background theories (e.g., integers or bit-vectors). The pre-condition, denoted by the *requires* clause in ACSL, characterises the legal inputs of a function, while the post-condition, denoted by the *ensures* clause, states properties of the function result, in relationship to the arguments. In addition to pre- and post-conditions, contracts in ACSL also commonly specify modified variables in an *assigns* clause, but this is only meaningful in the presence of global variables. An example of such a contract is shown in Listing 1.2.

A contract for a function f makes it possible to carry out the task of verifying a program (that uses f) in two steps:

(i) It has to be checked that the function f *satisfies* its contract. An implementation of a function f is said to satisfy a contract $(Pre_f, Post_f)$ if every terminating run that starts in a state satisfying Pre_f ends in a state satisfying $Post_f$. In Sect. 4, contract satisfaction will be formalised through the Hoare triple $\{Pre_f\}\ S_f\ \{Post_f\}$, in which S_f is the function body of f.

(ii) The rest of the program, sometimes called the *client code*, can be verified on the basis of the contract for f, disregarding the concrete implementation of f. A contract $(Pre_f, Post_f)$ is applicable in client code if every invocation of f satisfies the pre-conditions Pre_f, and if the client code executes correctly for every potential result of f satisfying $Post_f$. In Sect. 4, the verification of client code will be formalised through a dedicated Hoare proof rule Call.

In the case of a program with multiple functions, step (i) will be carried out for each implementation of a function in the program, so that the overall verification effort can be split into many small parts that can be handled separately or in parallel. As a result, verification of large programs can be organised in such a way that the verification effort scales roughly linearly in the size of the program, which is the key idea underlying procedure-modular (or function-modular) verification. All large-scale verification projects proceed in this modular manner.

2.2 Contract Refinement and Composition

A function implementation will generally satisfy many contracts, some of which will be sufficient to verify a given piece of client code, while others might be too weak. Different contracts might cover different aspects of function behaviour, and for instance describe the results produced for different input ranges. This motivates the study of algebraic properties of the space of contracts, an exercise that has received much attention for the case of system-level contracts [4], and to a lesser degree for contracts of functions or procedures [30]. We define notions of contract refinement, conjunction, and disjunction that exhibit several convenient properties, and that correspond to the way contracts are used in KeY [1].

We say that a contract $C = (Pre, Post)$ *refines* a contract $C' = (Pre', Post')$ (for the same function f), denoted $C \sqsubseteq C'$, if $Pre' \Rightarrow Pre$ and $Post \wedge Pre' \Rightarrow Post'$; in other words, refinement weakens the pre-condition of a contract, and

strengthens the post-condition for arguments admitted by the pre-condition. Refinement has the property that if a function f satisfies the finer (and thus *stronger*, or *more precise*) contract C, then f also fulfills the more abstract one C'. Vice versa, it also means that if a client verifies by means of modular verification against the more abstract contract C' of a function, it will again do so against the finer contract C, and thus does not need to be re-verified upon refining the contract. Finally, note that the conjunction $Post \wedge Pre'$ only makes sense under the assumption that function arguments (or global variables) cannot be updated in function bodies, as stated in the beginning of Sect. 2.1. In the more general case, Pre' has to be modified to refer to the function pre-state, for instance using the `\old` operator in ACSL.

The relation $\sqsubseteq$ is a preorder on the set of possible contracts of a function f, i.e., it is reflexive, transitive, but clearly not anti-symmetric. As usual, this means that the preorder induces an equivalence relation $\equiv$ defined by

$$C \equiv C' \;\Leftrightarrow\; (C \sqsubseteq C') \wedge (C' \sqsubseteq C)$$

and that the quotient $\sqsubseteq / \equiv$ is a partial order. In the following, we denote the class of contracts that are equivalent to C by $[C]$, but leave out the brackets $[\cdot]$ in most formulas for sake of presentation.

Example 2. Consider a function `abs` that computes the absolute value of an integer x. Possible contracts for `abs` are:

$$\begin{aligned} C_1 &: \; (x = 5,\, r = 5) & \qquad C_2 &: \; (x = 5,\, r = 6) \\ C_3 &: \; (x = 5,\, r = x) & \qquad C_4 &: \; (x \geq 0,\, r = x) \\ C_5 &: \; (x \leq 0,\, r = -x) & \qquad C_6 &: \; (\mathit{true},\, r = |x|) \end{aligned}$$

A standard implementation of `abs` will satisfy the contracts C_1, C_3, C_4, C_5, C_6, but not C_2. Contracts C_1 and C_3 are equivalent, $C_1 \equiv C_3$. Contract C_4 refines contract C_1 ($C_4 \sqsubseteq C_1$), since C_4 is more restrictive than C_1, and similarly $C_6 \sqsubseteq C_4$ and $C_6 \sqsubseteq C_5$.

The example indicates that the space of possible contracts of a function has the structure of a lattice, which is indeed the case:

Lemma 3. *Let $\mathcal{C}$ be the set of all contracts of a function f (including both satisfied and unsatisfied contracts). The partially ordered set $(\mathcal{C}/\equiv, \sqsubseteq / \equiv)$ is a bounded lattice with the bottom element $\bot = [(\mathit{true}, \mathit{false})]$, the top element $\top = [(\mathit{false}, \mathit{true})]$, and the binary operations:*

$$[(Pre, Post)] \sqcup [(Pre', Post')] \;\Leftrightarrow\; [(Pre \wedge Pre',\; Post \vee Post')] \qquad \text{(Join)}$$

$$[(Pre, Post)] \sqcap [(Pre', Post')] \;\Leftrightarrow\; [(Pre \vee Pre',\; (Pre \rightarrow Post) \wedge (Pre' \rightarrow Post'))] \qquad \text{(Meet)}$$

Proof. It mainly has to be verified that the defined operations indeed describe least upper and greatest lower bounds. This can be done automatically by first-order theorem provers. □

The bottom element $\bot$ of the contract lattice is the strongest (most refined) contract, and is only satisfied by an implementation of a function that diverges for every input; the top element $\top$ is the weakest contract, and satisfied by every implementation. Joining two contracts models the case of a function f having several implementations: if two implementations of f satisfy the contracts C_1 and C_2, respectively, then both implementations will satisfy $C_1 \sqcup C_2$, and client code can be verified on the basis of this more abstract contract. The meet of two contracts merges properties of a function expressed in multiple contracts into a single, stronger contract: if an implementation satisfies both C_1 and C_2, then it will also satisfy $C_1 \sqcap C_2$ (and vice versa). The meet operation is useful, in particular, for merging multiple automatically inferred contracts, as will be discussed later in this paper.

Example 4. We consider again the contracts from Example 2. Contracts C_4 and C_5 capture different parts of the behaviour of `abs`, and the meet of the two contracts is C_6, i.e., $C_4 \sqcap C_5 \equiv C_6$.

The problem of combining contracts has also been studied in the context of Hoare inference rules. Consider a library function that is called by several client functions. The library function may then have a different contract inferred from each client context, and we want to combine these into a single contract. Owe et. al. found that two inference rules, namely a rule for generalised normalisation and the Conseq rule of Hoare logic (see Table 1), are sufficient to infer any Hoare triple $\{P\}\ S\ \{R\}$ from a given set of Hoare triples over the same program S that logically entail $\{P\}\ S\ \{R\}$ [30]. Combining contracts in this way, however, does not preclude the need for re-verifying clients against their contracts in the context of the new, combined contract of a library function.

3 Existing Approaches in Contract Inference

While contracts are extremely useful for verifying programs modularly, finding correct and sufficient contracts is a time-consuming and error-prone process. This section surveys the existing work on *automatic* contract synthesis, extracting contracts either from the program under verification, or from the program together with the overall properties to be verified.

3.1 Strongest Post-conditions

One line of research on inferring contracts is based on the fundamental semantic notion of *strongest post-condition* due to Dijkstra [13], sometimes also called a *function summary*. Given a program (or rather a part of a program, typically a procedure) and a *pre-condition*, i.e., a state assertion that is assumed to hold at the beginning of the execution of the program, the strongest post-condition is an assertion that captures precisely the final states of the execution. So, if the pre-condition is *true*, the strongest post-condition characterises the final states of all executions starting in any initial state. For the purposes of contract

inference this is meaningful, since one obtains useful contracts even when no pre-condition has been supplied from elsewhere. The strongest post-condition provides the strongest possible contract (for the given pre-condition) and is thus ideal for procedure-modular deductive verification. However, this precision often comes at the cost of overly verbose assertions being generated. This is because the strongest post-condition, when generated automatically, will often closely reflect the program from which it is extracted, and not necessarily capture the program *intention* concisely.

Strongest post-conditions can be computed with the help of *symbolic execution* [15], at least for programs that do not contain unbounded loops. This method is a source of explosion of the size of the generated formulas, since it is path-based.

Singleton and others [36] developed an algorithm that converts the exponentially large post-conditions, resulting from a strongest post-condition computation, into a more concise and usable form. The algorithm consists of several steps. Initially, the program is converted into a passive single assignment form (essentially eliminating all assignments, which are the main source of assertion explosion). Then, from a given pre-condition (or *true*, if no better pre-condition is available), the strongest post-condition is computed by means of symbolic execution. The resulting formula is first converted into a normal form that groups subformulas into a hierarchy of cases, and is then flattened. The flattened formula is analysed for overlapping states, which are recombined into a form, in which the strongest post-condition is finally presented.

3.2 Weakest Pre-conditions

A related approach is the inference of *pre-conditions* that are sufficient to establish given post-conditions, or more generally pre-conditions that ensure that program assertions do not fail. The information captured by pre-conditions is complementary to that of post-conditions: while pre-conditions do not compare pre- and post-states, and therefore cannot specify the effect of a function, they state obligations required for correct execution of a function that need to be taken care of by the caller.

Pre-conditions can be computed in various different ways. The traditional weakest pre-condition calculus due to Dijkstra [12] transforms post-conditions to pre-conditions, and forms the basis of several deductive verification systems. Similarly to post-conditions, pre-conditions can also be computed through symbolic execution, by extracting the path constraints of all paths leading to violated post-conditions or failing assertions. An alternative approach based on a combination of abstract interpretation and quantifier elimination is presented by Moy [27]; the method is able to compute pre-conditions also in the presence of loops, but it does not always output weakest pre-conditions. Seghir and Kroening derived pre-conditions using an algorithm based on *Counterexample-Guided Abstraction Refinement* (CEGAR) [34]. Starting from an over-approximation of the weakest pre-conditions of a function, the algorithm iteratively eliminates

sets of pre-states for which function execution can lead to errors, until eventually sufficient (but still necessary, i.e., weakest) pre-conditions remain. The use of CEGAR implies that also functions with loops can be handled, although in general the algorithm might not terminate; this is unavoidable since neither strongest post-conditions nor weakest pre-conditions are computable in general. An eager variant of the algorithm, which does no longer require a refinement loop is demonstrated by Seghir and Schrammel [35].

The concept of *Maximal Specification Inference* [2] generalises the inference of weakest pre-conditions and considers the specifications of multiple functions simultaneously. Given a piece of program code that uses functions $f_1, \ldots, f_n$ (e.g., taken from a library), maximal specification inference attempts to construct the weakest specifications of $f_1, \ldots, f_n$ that are sufficient to verify the overall program. Albarghouthi et al. computed weakest specifications in a counterexample-guided manner, employing multi-abduction to find new specifications [2]. The approach can be applied even when no implementations of $f_1, \ldots, f_n$ are available.

3.3 Dynamic Inference of Assertions and Contracts

Given a set of test cases that exercise a system through its state space, a tool for *dynamic contract inference* will determine conditions that hold at various program points. These conditions are candidate assertions that may hold for all program runs. The most well-known tool for dynamic contract inference is Daikon [14], which automatically detects likely invariants for multiple languages; C, C++, Java and Perl. For each language, Daikon dedicates an instrumenter to trace certain variables. The traced variables are read by the inference engine to generate likely invariants. The generated invariants are tested against their trace samples and are reported only if they pass these tests. Daikon is optimised to handle large code sizes and large numbers of invariants, e.g., by suppressing the weaker invariants.

The Daikon tool has been used for dynamic contract inference in Eiffel. Polikarpova et al. compare, on 25 Eiffel classes, the programmer-provided contracts with automatically generated ones [32]. The results from this study show that a high portion of the inferred assertions were correct and relevant, that there were around 5 times more inferred assertion clauses than programmer-provided ones, and that only about 60% of the programmer-provided assertions were covered by the automatically inferred ones. The main conclusion from this work is that the inferred contracts can be used to correct and improve the human-written ones, but that contract inference can not completely replace the manual work. As the authors comment, this should not be surprising, since automatically inferred contracts are bound to document the behaviour of the program *as it is*, rather than document its *intent*.

A related approach is used by QuickSpec [7], a tool to automatically discover laws satisfied by functional programs. QuickSpec systematically enumerates possible terms and equations about functional programs, uses random testing to eliminate laws that do not hold, and applies a congruence graph data-structure

to eliminate more complicated laws that follow from known simpler equations. Like the other dynamic approaches, QuickSpec can sometimes erroneously propose laws that could not be ruled out based on the generated test cases, but experiments show that the approach performs well in practice, and is able to discover intricate facts about programs.

3.4 Property-Guided Contract Inference

The methods discussed so far are driven primarily by the considered program, and can be applied even if no specifications or properties are otherwise given. Property-guided contract inference methods, in contrast, do not attempt to find the most general contract satisfied by a function, but instead aim at discovering contracts that are just sufficient to verify some overall property of a program. The approach proposed in Sects. 4 and 5 of this paper, which is based on an encoding of programs and functions as *Constrained Horn Clauses*, falls into this category. Also weakest pre-condition methods, and in particular maximal specification synthesis [2], can start from existing specifications.

Since they do not aim for generality, property-guided methods can often find partial contracts that are more succinct than the complete contracts. For instance, by choosing stronger pre-conditions, it might be possible to find contracts that ignore complicated corner-cases, provided that they are never triggered by a program.

As a related approach, Denney and Fischer propose the use of syntactic patterns to infer annotations for automatically generated code [11]. This is feasible since generated code is very idiomatic, so that all code constructs that can possibly occur can be covered by patterns. The chosen code generator systems were AutoBayes and AutoFilter. The results show that the algorithm could successfully certify various safety properties of the generated code.

4 Deductive Verification and Horn Clauses

This part discusses the property-guided inference of contracts with the help of Horn solvers. Horn clauses, in this context often narrowed down to *Constrained Horn Clauses*, CHC, have been proposed as a uniform framework to automate the application of proof rules in deductive verification [6,16,20], lifting deductive approaches to a similar level of automation as achieved by software model checkers. Thanks to their generality, Horn solvers can be applied quite naturally to infer not only program invariants, but also contracts, from the program and properties to be verified. We start by considering different calculi for deductive verification, and their automation, before more formally defining the framework of Horn clauses.

4.1 From Hoare Logic to Horn Clauses, and Back Again

Program logics characterise the correctness of programs in terms of proof rules that can be used to derive program correctness judgements. The proof rules often

require the right annotations to be provided by the user, or by some oracle; for instance, loop rules need (inductive) loop invariants to be provided, and rules for modular handling of function or method calls rely on a contract. We consider proof rules in Hoare logic [18], which express the correctness of programs in terms of Hoare triples $\{P\}\ S\ \{R\}$ with a pre-condition P, program S, and post-condition R. The triple $\{P\}\ S\ \{R\}$ expresses that every terminating run of program S that starts in a state satisfying P ends in a state satisfying R. Hoare triples can be derived using proof rules such as the ones in Table 1 and 2.

Table 1. Selection of standard hoare rules for sequential programs

$$\frac{}{\{P[x/t]\}\ x = t\ \{P\}}\ \textsc{Assign} \qquad \frac{\{P\}\ S\ \{Q\} \qquad \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}\ \textsc{Comp}$$

$$\frac{\{P \wedge B\}\ S\ \{R\} \qquad \{P \wedge \neg B\}\ T\ \{R\}}{\{P\}\ \mathit{if}\ B\ \mathit{then}\ S\ \mathit{else}\ T\ \{R\}}\ \textsc{Cond} \qquad \frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \mathit{while}\ B\ \mathit{do}\ S\ \{I \wedge \neg B\}}\ \textsc{Loop}$$

$$\frac{P \Rightarrow P' \qquad \{P'\}\ S\ \{R'\} \qquad R' \Rightarrow R}{\{P\}\ S\ \{R\}}\ \textsc{Conseq}$$

Sequential Hoare Proofs. The first selection of Hoare rules, shown in Table 1, can be used to derive properties of sequential programs without function calls. For simplicity, we assume that no distinction is made between program expressions and terms of the specification language in rule Assign, and between Boolean program expressions and formulas of the specification language in the rules Cond and Loop. One rule exists for each of the program constructs, which are assignments to variables, sequential composition, conditional statements, and while loops; one further rule, the Conseq uence rule, describes how pre-conditions can be strengthened and post-conditions be weakened. Each rule specifies that the conclusion, the Hoare triple underneath the bar, follows from the premises above the bar. In Assign, the notation $P[x/t]$ denotes (capture-avoiding) substitution of all free occurrences of x in P by the term t.

During proof search, the rules are applied backwards, in a goal-directed manner. Two of the rules have formulas in their premises that do not appear in the respective conclusion; when these rules are applied backwards, these formulas need to be provided by the user: in Comp, an *intermediate assertion* Q is needed to decompose a Hoare triple into two triples, and in Loop, a *loop invariant* I has to be specified. The framework of Horn clauses provides a general strategy to compute such verification artefacts automatically, by initially keeping the required annotations *symbolic* (i.e., by using uninterpreted predicate symbols), collecting constraints from the leaves of a proof tree, and then using a Horn solver to determine which annotations are adequate.

To implement this strategy, it is advantageous to work with slightly generalised versions of some of the rules, shown in Table 2. Compared to the rules

Table 2. Hoare rules with generalised conclusions

$$\frac{P \Rightarrow R[x/t]}{\{P\}\ x = t\ \{R\}}\ \textsc{Assign}' \qquad \frac{P \Rightarrow I \qquad \{I \wedge B\}\ S\ \{I\} \qquad I \wedge \neg B \Rightarrow R}{\{P\}\ \mathit{while}\ B\ \mathit{do}\ S\ \{R\}}\ \textsc{Loop}'$$

in Table 1, the generalised rules do not syntactically restrict the pre- and post-conditions in the conclusion; logically, the rules in Table 2 can be derived from their respective original version in Table 1 and the Conseq rule.

$$\dfrac{\dfrac{I(n,x) \wedge x < n \Rightarrow I(n,x+1)}{\{I(n,x) \wedge x < n\}\ x = x+1\ \{I(n,x)\}}}{\mathcal{P}}$$

$$\dfrac{\dfrac{n \geq 0 \Rightarrow P(n,0)}{\{n \geq 0\}\ x = 0\ \{P(n,x)\}} \qquad \dfrac{P(n,x) \Rightarrow I(n,x) \qquad \mathcal{P} \qquad I(n,x) \wedge x \not< n \Rightarrow x = n}{\{P(n,x)\}\ \mathit{while}\ x < n\ \mathit{do}\ x = x+1\ \{x = n\}}}{\{n \geq 0\}\ x = 0; \mathit{while}\ x < n\ \mathit{do}\ x = x+1\ \{x = n\}}$$

Fig. 1. Proof for Example 5

Example 5. We show how to prove a simple Hoare triple using this approach:

$$\{n \geq 0\}\ x = 0; \mathit{while}\ x < n\ \mathit{do}\ x = x+1\ \{x = n\}$$

A completely expanded proof tree for this Hoare triple is shown in Fig. 1. The proof contains applications of the rules Comp and Loop′, both of which demand program annotations; to be able to construct a complete proof, *symbolic formulas* $P(n,x)$ and $I(n,x)$ have been inserted, involving the uninterpreted binary predicate symbols P and I applied over the program variables, allowing us to postpone the actual choice of concrete formulas at this point.

The proof's leaves, marked in grey, represent the conditions that the formulas $P(n,x)$, $I(n,x)$ have to satisfy in order to close the proof:

$$\begin{aligned} n \geq 0 &\Rightarrow P(n,0) \\ P(n,x) &\Rightarrow I(n,x) \\ I(n,x) \wedge x < n &\Rightarrow I(n,x+1) \\ I(n,x) \wedge x \not< n &\Rightarrow x = n \end{aligned}$$

The variables n, x are implicitly universally quantified in each formula. The four conditions can be turned into *Horn clauses*, i.e., written as disjunctions with at most one positive literal each (Example 7 elaborates on this, continuing the current example), and their satisfiability can therefore be checked automatically

using Horn solvers. In case of our four conditions, a Horn solver would quickly determine that the clauses are indeed satisfiable, and that one possible solution are the formulas:

$$\begin{aligned} P(n,x) &\equiv n \geq 0 \wedge x = 0 \\ I(n,x) &\equiv n \geq x \wedge x \geq 0 \end{aligned}$$

To obtain a self-contained Hoare proof, we could substitute the placeholders P, I in Fig. 1 with those formulas, and observe that indeed all rule applications become valid, and the proof is well-formed.

Verification by Contract. One reason for the popularity of Horn clauses in verification is that other language features, for instance procedure calls or concurrency, can be handled in much the same way as sequential programs. We consider the case of a program containing (possibly mutually recursive) functions $f_1, \ldots, f_n$, and make the same simplifying assumptions as in Sect. 2: function parameters are passed by value and are read-only, and there are no global variables. For simplicity of presentation, it is assumed here that each function f is associated with a distinct set $\bar{a}_f = \langle a_f^1, \ldots, a_f^k \rangle$ of variables representing the formal arguments of the function, as well as a further distinct variable r_f to store the function result. Each function f is implemented through a function body S_f, which by itself is a piece of program code, and possibly contains function calls.

Table 3. Hoare rule for function calls

$$\frac{P \Rightarrow \mathit{Pre}_f[\bar{a}_f/\bar{t}] \qquad P \wedge \mathit{Post}_f[\bar{a}_f/\bar{t}] \Rightarrow R[x/r_f]}{\{P\}\ x = f(\bar{t})\ \{R\}}\ \textsc{Call}$$

Following the style of *design-by-contract* [26], each function f is specified with a contract $(\mathit{Pre}_f, \mathit{Post}_f)$, containing a pre-condition Pre_f over the arguments $\bar{a}_f$, and a post-condition Post_f over the arguments $\bar{a}_f$ and the result r_f. To verify programs involving function calls, we need a further Hoare rule, which is shown in Table 3. The rule enables the modular verification of programs referring entirely to the function contracts, and is a simplified version of the rules that can be found in the literature (e.g., in the work by von Oheimb [29]). The two premises of the rule state that function calls have to establish the pre-conditions, and that the post-conditions can be assumed to hold for the result of the function call. To verify an end-to-end property $\{P\}\ S\ \{R\}$ of a program S, with functions $f_1, \ldots, f_n$, in a procedure-modular way, it has to be shown that (i) each function f_i satisfies its contract, which means that the Hoare triple $\{\mathit{Pre}_{f_i}\}\ S_{f_i}\ \{\mathit{Post}_{f_i}\}$ holds; and (ii) S satisfies the end-to-end property $\{P\}\ S\ \{R\}$, making use of the contracts for the functions $f_1, \ldots, f_n$.

To verify programs with function calls automatically, we can apply a similar strategy as before: we keep function pre- and post-conditions initially symbolic

as formulas $Pre_f(\bar{a}_f)$ and $Post_f(\bar{a}_f, r_f)$, respectively; we collect the constraints that pre- and post-conditions have to satisfy, together with constraints on loop invariants and intermediate assertions; and finally we use a Horn solver to search for a solution of the constraints in combination. Note that we overload here the meta-symbols Pre_f and $Post_f$ used in rule CALL, with the uninterpreted predicate symbols of the symbolic formulas. Note also that while the formulas are symbolic, the terms to which the predicate symbols are applied are explicit, and thus the substitutions are carried out immediately (and not left symbolic).

Example 6. We show how to verify a program with a recursive unary function f using this strategy:

$$\{x \geq 0\}\ y = f(x)\ \{y = x\} \tag{1}$$

where the function f has the body

$$\mathit{if}\ a_f^1 > 0\ \mathit{then}\ \ z = f(a_f^1 - 1); r_f = z + 1\ \ \mathit{else}\ \ r_f = 0$$

Note that the notation $r_f = t$ corresponds to a *return* statement.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x \geq 0 \Rightarrow Pre(x) \qquad x \geq 0 \wedge Po(x, r_f) \Rightarrow r_f = x}{\{x \geq 0\}\ y = f(x)\ \{y = x\}}}{\dfrac{Pre(a_f^1) \wedge a_f^1 > 0 \Rightarrow Pre(a_f^1 - 1) \qquad Pre(a_f^1) \wedge a_f^1 > 0 \wedge Po(a_f^1 - 1, r_f) \Rightarrow Q(a_f^1, r_f, r_f)}{\{Pre(a_f^1) \wedge a_f^1 > 0\}\ z = f(a_f^1 - 1)\ \{Q(a_f^1, z, r_f)\}} \qquad \dfrac{Q(a_f^1, z, r_f) \Rightarrow Po(a_f^1, z + 1)}{\{Q(a_f^1, z, r_f)\}\ r_f = z + 1\ \{Po(a_f^1, r_f)\}}}{\{Pre(a_f^1) \wedge a_f^1 > 0\}\ z = f(a_f^1 - 1); r_f = z + 1\ \{Po(a_f^1, r_f)\}}}{\mathcal{P}_1} \qquad \dfrac{\dfrac{Pre(a_f^1) \wedge a_f^1 \not> 0 \Rightarrow Po(a_f^1, 0)}{\{Pre(a_f^1) \wedge a_f^1 \not> 0\}\ r_f = 0\ \{Po(a_f^1, r_f)\}}}{\mathcal{P}_2}}{\{Pre(a_f^1)\}\ \mathit{if}\ a_f^1 > 0\ \mathit{then}\ z = f(a_f^1 - 1); r_f = z + 1\ \mathit{else}\ r_f = 0\ \{Po(a_f^1, r_f)\}}$$

Fig. 2. Hoare proofs for Example 6

The two proof trees needed to verify the program, for the overall property and the correctness of the function contract, are shown in Fig. 2, using the symbolic formulas $Pre(a_f^1)$ and $Po(a_f^1, r_f)$ for the contract and $Q(a_f^1, z, r_f)$ as intermediate assertion between $z = f(a_f^1 - 1)$ and $r_f = z + 1$. The proofs give

rise to the following conditions (in the proofs in grey) about the annotations:

$$\begin{aligned} x \geq 0 &\Rightarrow \mathit{Pre}(x) \\ \mathit{Pre}(a_f^1) \wedge a_f^1 > 0 &\Rightarrow \mathit{Pre}(a_f^1 - 1) \\ \mathit{Pre}(a_f^1) \wedge a_f^1 > 0 \wedge \mathit{Po}(a_f^1 - 1, r_f) &\Rightarrow Q(a_f^1, r_f, r_f) \\ Q(a_f^1, z, r_f) &\Rightarrow \mathit{Po}(a_f^1, z + 1) \\ \mathit{Pre}(a_f^1) \wedge a_f^1 \not> 0 &\Rightarrow \mathit{Po}(a_f^1, 0) \\ x \geq 0 \wedge \mathit{Po}(x, r_f) &\Rightarrow r_f = x \end{aligned}$$

A solution of the constraints is:

$$\begin{aligned} \mathit{Pre}(a_f^1) &\equiv \mathit{true} \\ \mathit{Po}(a_f^1, r_f) &\equiv (a_f^1 \geq 0 \wedge r_f = a_f^1) \vee (a_f^1 \leq 0 \wedge r_f = 0) \\ Q(a_f^1, z, r_f) &\equiv z = r_f \wedge a_f^1 = z + 1 \wedge a_f^1 > 0 \end{aligned}$$

In other words, it has been shown that the function f satisfies the contract $C = (\mathit{true}, (a_f^1 \geq 0 \wedge r_f = a_f^1) \vee (a_f^1 \leq 0 \wedge r_f = 0))$, and that C is sufficient to verify the client program (1). It can be noted that C, for reasons of readability, can be decomposed into $C \equiv C_1 \sqcap C_2$, with simpler contracts $C_1 = (a_f^1 \geq 0, r_f = a_f^1)$ and $C_2 = (a_f^1 \leq 0, r_f = 0)$.

4.2 Constrained Horn Clauses

We now introduce the framework of constrained Horn clauses more formally. Throughout the section, we assume that some background theory has been fixed, for instance the theory of Presburger arithmetic, of fixed-length bit-vectors, or of arrays. Given a set X of first-order variables, a *constraint language* is then a set *Constr* of first-order formulas over the background theory and X; in practice, often the constraint language is restricted to quantifier-free formulas.

We then consider a set R of uninterpreted fixed-arity relation symbols, which represent set-theoretic relations over the domain described by the background theory. Relation symbols are used as symbolic formulas (e.g., I, Pre, etc.) in the previous section.

A *(constrained) Horn clause* is a formula $B_1 \wedge \cdots \wedge B_n \wedge C \rightarrow H$ where

- $C \in \mathit{Constr}$ is a constraint over the chosen background theory and X;
- each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in R$ to first-order terms over X;
- H is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in R$ to first-order terms, or *false*.

The first-order variables in a clause are implicitly universally quantified. H is called the *head* of the clause, and $B_1 \wedge \cdots \wedge B_n \wedge C$ the *body*. In case $C = \mathit{true}$, we often leave out C and just write $B_1 \wedge \cdots \wedge B_n \rightarrow H$.

Horn solvers are tools for computing (symbolic or syntactic) solutions of Horn clauses. A syntactic solution of a set HC of Horn clauses maps every relation symbol $p(x_1, \ldots, x_n)$ (with $p \in R$) to a constraint over the arguments $x_1, \ldots, x_n$ in the chosen constraint language, in such a way that substituting the formulas for the relation symbols makes all clauses valid. Tools like Spacer [25] or Eldarica [19] can compute solutions of Horn clauses over a number of background theories fully automatically, with the help of model checking algorithms including CEGAR and IC3.

Example 7. The four conditions extracted in Example 5 can be turned into Horn clauses with just a minor change of notation. The clauses are formulated over the set $R = \{P, I\}$ of binary relation symbols, and the first-order variables n, x are implicitly universally quantified:

$$\begin{aligned} n \geq 0 &\rightarrow P(n, 0) \\ P(n, x) &\rightarrow I(n, x) \\ I(n, x) \wedge x < n &\rightarrow I(n, x + 1) \\ I(n, x) \wedge x \not< n \wedge x \neq n &\rightarrow \mathit{false} \end{aligned}$$

It can be noted that the formulas given in Example 5 are indeed a syntactic solution of the clauses.

4.3 Tool Support

We conclude the section by discussing the model checkers and deductive verification tools used in the rest of the paper.

TriCera. There are several verification tools implementing the verification strategy outlined in Sect. 4.1, including the tool SeaHorn [17] for C programs, and JayHorn [23] for Java. TriCera,[1] the tool used in our experiments, is a software model checker for C programs following the same methodology. TriCera was originally a spin-off of the C front-end that was used in the Horn solver Eldarica, and later extended to also support computation of function contracts, and to handle heap-allocated data-structures. TriCera primarily targets the Horn solver Eldarica as back-end, but can also output Horn clauses to interface other solvers.

Eldarica. Eldarica [19][2] is a solver for Horn clauses with constraints over a number of possible theories. Eldarica combines Predicate Abstraction with the Counterexample-Guided Abstraction Refinement (CEGAR) algorithm to automatically check whether a given set of Horn clauses is satisfiable. Eldarica first appeared as a solver for Horn clauses over Presburger arithmetic in 2013. Over the last years, various further features have been added to the tool, and it

[1] https://github.com/uuverifiers/tricera.
[2] https://github.com/uuverifiers/eldarica.

can now solve problems over the theories of integers, algebraic data-types, bit-vectors, and arrays. Eldarica can process Horn clauses and programs in a variety of formats, implements sophisticated heuristics to solve tricky systems of clauses without diverging, and offers an elegant API for programmatic use.

Frama-C. Frama-C is a tool for static analysis of C code [9]. There are several plugins available that perform different analyses. On such plugin is WP, which uses deductive verification to verify functional properties of programs [9]. To perform verification Frama-C uses function contracts to specify behaviours of functions, and to this end it has its own specification language called ACSL (ANSI C Specification Language). Function contracts are annotated in the source code as a special type of C comments, and a contract generally consists of pre-conditions and post-conditions, expressed using the keywords `requires` and `ensures`, respectively. An example can be seen in Listing 1.2.

The deductive verification performed by the WP plugin is based on Weakest Pre-condition calculus [12], which defines for a statement S a function from any post-condition Q to a pre-condition P', such that P' is the weakest pre-condition where the Hoare triple $\{P'\}\ S\ \{R\}$ holds. This mapping can be seen as a modification of the Hoare logic rules presented in Sect. 4.1, and verification of a Hoare triple $\{P\}\ S\ \{R\}$ can then be performed by computing the weakest pre-condition P' and proving that $P \Rightarrow P'$. The WP plugin verifies, and reports, for each annotation (e.g., assertions and invariants) whether it is valid. The overall program is said to be valid if all annotations are valid. The verification process is function-modular, i.e., when a function call is reached the tool checks that the caller fulfills the pre-condition of the called function, and, if so, assumes that the post-condition hold after the function. Verification that the callee fulfills the contract is then performed separately, similarily to the proof in Fig. 2.

5 Synthesising Contracts for Frama-C

This section describes how the model checker TriCera is used to synthesise contracts that can be verified in Frama-C, and the reason for doing so.

The way TriCera is used to synthesise contracts is by verifying that all assertions in the program hold. As a side effect of this, contracts for all individual functions are generated, as part of the Horn clause solution. Through an extension to TriCera, these contracts are then syntactically transformed into ACSL, and verified in Frama-C. Using a deductive verifier to verify the program again, with the newly generated contracts, increases confidence in the verification result. Frama-C is a well established tool that is generally considered trustworthy, and using several verification techniques naturally increases confidence.

Extracting contracts from model checking can also help in other respects. Contracts can be used when a program is modified to speed up the re-verification process: all contracts that are still satisfied by a modified function can be kept and reused. Contracts can also be applied to gradually modularise verification efforts, and in this way improve scalability. Since model checking is automatic but

```
mc914/2: (((_1 + −1 ∗ _0) = 0) & ((100 + −1 ∗ _0) >= 0))
mc913/2: (((_1 + −1 ∗ _0) = 0) & ((−101 + _0) >= 0))
 ...
mc91_pre/1: true
 ...
mc91_post/2: (((!((10 + (_1 + −1 ∗ _0)) = 0) | ((−101 + _0) >= 0)) & (!((−91
      + _1) = 0) | ((101 + −1 ∗ _0) >= 0))) & (((10 + (_1 + −1 ∗ _0)) = 0) |
      ((−91 + _1) = 0)))
```

Listing 5.1. A portion of TriCera's solution output for Listing 1.1.

not function modular, it is by itself limited in scalability. By relying mainly on the result of Frama-C, contract synthesis is complementary to writing contracts manually. Modules that are too large to be model-checked can still be verified and the result combined with verification of other modules, where all contracts have been synthesised automatically. Finally, having explicit contracts is useful for verification of code external to the code base used to synthesise the contracts, for example when the latter is a library.

In the present paper, we assume a context where there are properties to be verified at the C module level. Even though the contracts do not give a complete specification of function, they are still of interest in this context, since they will always be sufficient to prove the desired properties at the module level.

TriCera does currently not support the full C language, and only has limited support for heap, arrays, pointers, and structs. As such, properties related to memory safety cannot be verified, and contracts not be generated. We therefore limited our experiment to programs over arithmetic data-types.

5.1 Syntactical Transformation

Our starting point for producing ACSL contracts is the Horn clause solution output in Prolog format by TriCera. A part of the Horn clause solution for the example in Listing 1.1 is shown in Listing 5.1. Each line of the output contains a formula that defines one of the predicates from the Horn clauses.

In TriCera's solution, the pre-state prior to entering the function, and the post-state after exiting the function are identified with the names `func_pre` and `func_post`. Those were split from other statements such as the main invariant and the conditions of functions at different states. The pre-state and post-state formulas correspond to the pre-condition and post-condition, i.e, the function contract. After extracting the contracts, they were properly rearranged twice. The first rearrangement was with respect to the function name. The second rearrangement was required to ensure that the pre-condition precedes the post-condition of each function. Thus, the order is compliant with ACSL annotations.

```
/*@
//Function: mc91
    requires \true;
    ensures (\result  − \old(n) != −10 || \old(n) >= 101) &&
            (\result  != 91  ||  101 >= \old(n)) &&
            (\result  − \old(n) == −10 || \result == 91);
*/
```

Listing 5.2. The final result of pretty-printing an ACSL contract, when using the harness from Listing 1.1

TriCera uses the symbols `_0`,`_1`, ... `_n-1` to represent to index function arguments and program variables. For outputting contracts, those symbols were again replaced with the original program variables. A few more syntactical modifications were necessary in order to adhere to the ACSL format, as follows:

(i) Logical symbols (`&`, `|`, `=`) were replaced with the C equivalents (`&&`, `||`, `==`).
(ii) Boolean literals (`true`, `false`) were replaced with the corresponding ACSL primitives (`\true`, `\false`).
(iii) Variable values before execution (`value_old`) were replaced with the equivalent ACSL construct (`\old(value)`).
(iv) The keywords for pre- and post-states (`func_pre`, `func_post`) were replaced with ACSL keywords for pre- and post-conditions (`requires`, `ensures`).

To form more readable contracts, we used (and extended) the existing pretty-printer from Princess [33], the theorem prover included in TriCera and Eldarica. Pretty-printing eliminated most of the parentheses in the output, and applies further simplifications to the formulas, leading to more legible ACSL. The final result is shown in Listing 5.2. With some manipulations of the formula it is easy to see that this is equivalent to the contract in Listing 1.2. Furthermore, we can see that the contract is a complete specification of the function, despite the input specification not asserting properties for all executions.

5.2 Contracts for Different Assertions

In the generation of ACSL contracts, the result highly depends on the choice of the logical formula in the assertions. For example, in Listing 1.1, if the harness is altered to only have the assertion `assert(res>0);`, then the generated post-condition is `ensures \result >= 91;`, and if altering the harness to have only the assertion `assert(res==91 || res==x-10);`, the generated post-condition is `ensures \result == 91 || (\result-\old(n)==-10 && \old(n)>=101);` (in both cases the pre-condition is simply `true`). Both post-conditions are verified in Frama-C as `valid`, and proven to be correct with respect to the test harness.

There is no guarantee of completeness of the generated contract, it will simply be sufficient to verify what was asserted about the function, as well as be sound with regard to recursive calls. However, in many cases, such as in the example just shown in Listing 5.2, the contract will be stronger than what was asserted in the harness function.

5.3 Contracts for Client Code and Libraries

As explained above, the function contracts that are inferred are sufficient for the calling function to be verified, but are not necessarily the *strongest* (or *most precise*) contracts that the respective functions fulfill. As such, if a function is used in more than one client program (say it is a library function), the contract extracted in the context of one client might not be sufficient in the context of another client. If we have several clients using the same library function, the contracts inferred individually in each client can be combined to a single contract that allows all client code to be verified.

Depending on the context in which a function is called, the pre-condition generated by TriCera will vary. For example, if, as in Listing 1.1, the function inputs are unconstrained, the pre-condition generated tends to be `\true`. This is because the function is in fact model-checked for all possible states w.r.t. the affected variables. In this case, the generated contracts can be combined by simply creating a new post-condition that is the conjunction of all generated post-conditions. In Frama-C this can be achieved by including all the generated `ensures` clauses, since this is semantically equivalent to having one `ensures` clause that is the conjunction of all the expressions. The new contract C will then refine all the generated contracts $C_1, \ldots, C_n$, as defined in Sect. 2.2.

A more interesting case is when a function is called in a context where the variables are assigned a specific value, or their possible values are a subset of the total range. In this case TriCera will generate a pre-condition allowing only the values that can possibly occur in the calling context. For example, consider the `cmp` function seen in Listing 5.3, which is used as an example in the paper by Singleton et al. [36] discussed in Sect. 3.1. For some harness, TriCera will generate the clause `requires b == 5 && a == 5;` as pre-condition, with the post-condition `ensures \result == 0 && \old(b) == 5 && \old(a) == 5;`, and for some other harness the two clauses `requires b == 7 && a == 5;` and `ensures \result == -1 && \old(b) == 7 && \old(a) == 5;` will be generated. Note that the post-conditions contain redundant equalities over the variables in the pre-state, an artefact of the use of constant propagation in the Horn solver. In this case we cannot use conjunctions of the contracts to create a new contract. Ignoring the fact that the post-conditions would be incompatible, the specification resulting from contract conjunction would not give enough information to the caller about which input would create the respective output. Instead one can use each pair of pre- and post-conditions (P_i, Q_i) to form an implication $P_i \Rightarrow Q_i$, to create a new post-condition that is the conjunction of all these implications. This follows the meet operation defined in Sect. 2.2, and

```
int cmp(int a, int b) {
    int c = a;
    if (c < b) {
        return −1;
    } else {
        if (c > b) {
            return 1;
        }
        return 0;
    }
}
```

Listing 5.3. C implementation of an integer comparison function.

thus we have that the new contract $C = C_1 \sqcap \cdots \sqcap C_n$, where $C_1, \ldots, C_n$ are the generated contracts.

An equivalent result can also be achieved by using the ACSL construct `behavior`. Behaviours are used specifically to specify several pre- and post-condition pairs, which is also evaluated similarly to the meet operation on contracts. The two approaches are semantically equivalent, and as long as we also keep the disjunction of pre-conditions will have no effect on verification completeness. Without the pre-condition, the contract might not be possible to prove. An obvious example is when the pre-condition contains auxiliary assertions, for example about memory validity, since then the resulting specification might not be possible to prove because of run-time exceptions not related to executions considered in the particular contexts from which the contracts were generated.

Listing 5.4 shows an example of three automatically inferred contracts that have been manually conjoined using the Frama-C `behavior` construct as outlined above, into a single contract that can be verified in Frama-C, and which allows the clients from which the original, now conjoined, contracts where generated to be verified. By using this approach, it also possible to instruct Frama-C to prove that the behaviours are disjoint, and that they form a complete specification, if desired, by using the ACSL keywords `disjoint` and `complete`. Completeness means that the assumptions of the behaviours covers all possible states as specified by the pre-condition, i.e. that the pre-condition implies that at least one of the assumptions of the behaviours hold. Disjointness refers to the assumptions of the behaviours not overlapping, i.e. that the pre-condition implies that no two assumptions of the behaviours hold at the same time.

6 A Case Study Using SV-COMP Benchmarks

This section describes how the TriCera contract generation was evaluated using SV-COMP verification tasks, and the results thereof. The case study is a

```
/*@
    requires a == b || a < b || a > b;
    assigns \nothing;
    behavior eq:
        assumes b == a;
        ensures \result == 0 && \old(b) == \old(a);
    behavior lt:
        assumes b − a >= 1;
        ensures \result == −1;
    behavior gt:
        assumes a − b >= 1;
        ensures \result == 1;
    complete behaviors;
    disjoint behaviors;
*/
```

Listing 5.4. Frama-C contract resulting from the conjoining of different inferred contracts for the function in Listing 5.3.

continuation of previous work on using model checking based on Constrained Horn Clauses to verify and infer contracts for industrial software [3].

The authors of the present paper are not aware of any existing techniques for C code contract inference, and cannot therefore make a comparative evaluation. Instead, a subset of a collection of verification tasks commonly used to evaluate verification techniques was used to carry out initial experiments with the contract inference performed by TriCera.

6.1 SV-COMP Verification Tasks

The International Competition on Software Verification (SV-COMP) is an annual competition to assess the state-of-the-art software verification tools [5]. The collection of verification tasks used in this competition is maintained in an open-source repository, and has been contributed by multiple research and development groups [37].

The SV-COMP repository was chosen as the main method to test the automatically generated contracts. The benchmark suite was limited to 12 folders of C implementation files. The selected verification tasks focused on checking loops and recursions. The tested properties varied in nature, and included, for example, overflow and (un)reachability checks.

It was necessary to edit the source files to prepare them for contract generation, and also to process them correctly with TriCera. The main changes included:

– Some source files included loops directly inside a main function, but no function calls. In such files the loops were moved to separate functions, and

contracts generated for those auxiliary functions. New variables were introduced to store the returned values, and used in the properties to be verified in the outer function.
- Tasks with the expected verification result `FALSE`, i.e., a counterexample to safety, were modified to produce the answer `TRUE`.
- Some source files used the function `VERIFER_error()` to express assertions. Such function calls were changed to the statement `assert(0)`.
- The reserved keyword `_Bool` is not supported by TriCera. Source files using this data-type were fixed by adding a `typedef enum {false, true} _Bool;`

The experiments were performed in a virtual machine running Ubuntu 17.10, on a host machine with an Intel Core i5-7500 CPU @ 3.40 GHz. Three of the four CPU cores, and 3.8 GiB memory, were allocated to the virtual machine. The verification time limit was set to 60 min. Where we report average contract generation times, the test cases that timed out were excluded.

6.2 Results

The following verification results focus on whether functions meet their generated contracts, so the result is considered to be positive (i.e., verified) when the post-condition is verified as `Valid` using Frama-C. Some results also specify whether the pre-condition was verified as `Valid`, which means that it holds at all call sites (including both the harness function and recursive calls). A result of `Unknown` means that Frama-C terminated but was unable to prove the assertion, and `Timeout` means that the verification attempt did not terminate within the time limit.

Overall Results. In total there were 129 verification tasks tested using TriCera. For 110 of these, a contract could be generated within the set time limit, whereas 19 tasks timed out. Out of the 110 generated contracts, 78 could be immediately verified by Frama-C. An additional 20 tasks could be verified after manually adding loop annotations (i.e., `variant`, `invariant` and `assigns`). The rest of the verification tasks could not be verified. The average contract generation time was 19 s, with the minimum and maximum being 1.3 s and 17 min, respectively. The detailed results are divided into two parts based on whether the files contained functions with loops or recursion.

Programs with Loops. This part of the experiment was conducted over 66 source files, which were selected from 10 different folders of the SV-COMP benchmark repository. The test suite contained both `For` and `While` loops, and also some cases of nested loops. Programs had between 10 and 50 lines of code (measured using the tool CLOC [10]).

Of the 66 programs with loops, 13 did not have a contract generated within the time limit, and were ignored for the rest of the analysis. Of the 53 for which a contract was generated, 22 could immediately be verified with Frama-C, in addition to 20 more verified after manually adding loop annotations. The primary

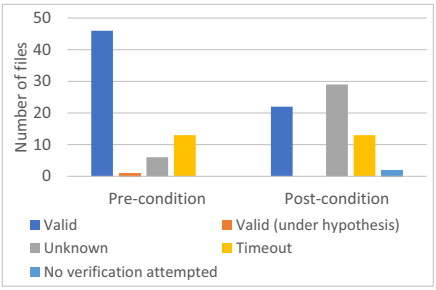

Fig. 3. Results of verifying generated contracts for files containing loops in Frama-C.

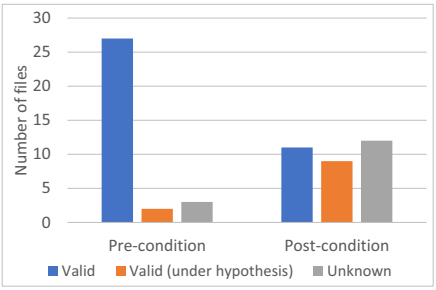

Fig. 4. Results of verifying generated contracts for files with loops in Frama-C, after adding loop annotations.

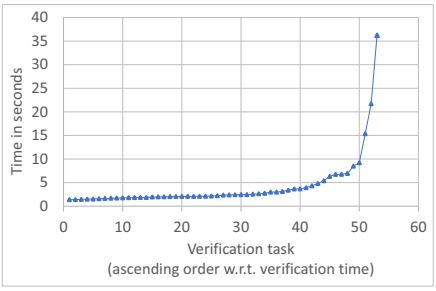

Fig. 5. Time spent in running both verification and contract generation in TriCera of functions that contain loops.

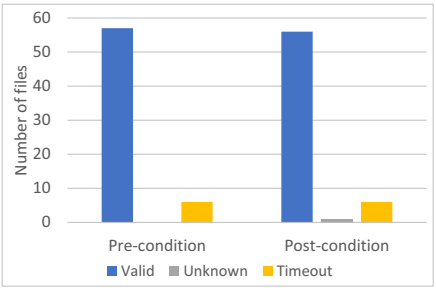

Fig. 6. Results of verifying generated contracts for files with recursive functions in Frama-C.

verification result is shown in Fig. 3 in terms of pre-condition and post-condition verification status in Frama-C. In Fig. 4, the verification result after manually adding loop annotations can be seen, for the files which were not already successfully verified. In several of these cases, Frama-C verified the assertions as valid under some hypothesis, which means that the result depended on other assertions that could not be verified. In almost all cases, this dependence were the manually added loop annotations that could not be verified.

The average contract generation time was 4.2 s, the minimum 1.4 s, and the maximum 36 s (Fig. 5). The horizontal axis views a series of verification tasks, while the vertical axis shows the ascending order of the required time.

Programs with Recursion. This part of the experiment was conducted over 63 source files that contained recursive functions, selected from 2 folders of the SV-COMP benchmark repository. These files all contained a main function, which could be used as a harness, calling recursive functions. Thus, the source code did not require to be rewritten in the same manner as the files containing loops. The source code included both single and nested recursive (up to 4) functions. Programs had between 18 and 66 lines of code.

Of the 63 programs with recursion, 6 failed the contract generation time limit. Of the 57 programs with generated contracts, 56 could be verified with Frama-C. Figure 6 shows the final result of the contracts tested using Frama-C. No further additions were required to verify those files.

The average contract generation time was 33 s, with the minimum 1.3 s and maximum 17 min (Fig. 7).

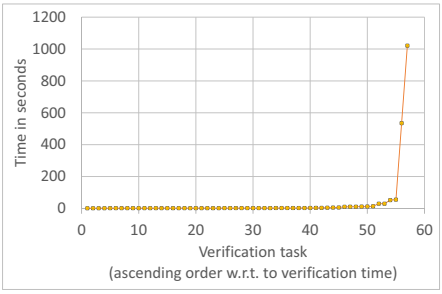

Fig. 7. Time spent in running both verification and contract generation in TriCera of functions that contain recursion.

Summary. The case study used the SV-COMP repository as the source to test the inferred contracts. There were no further assertions added to the source files, and only 5 files required assumption statements in order to generate the contracts (e.g. to ensure that a random value assigned to a loop guard variable is always positive). Around 77.3% of files with loops (excluding the time outs) and 98.2% of files with recursion were verified with Frama-C. The overall time varied from a few seconds to 17 min.

The tasks with loops demanded extensive manual work because of the missing `invariant`, `variant` and `assigns` loop annotations. However, the recursive tasks were straightforward, and minimal code alterations were made.

7 Conclusions

This paper surveys existing work on automatic inference of program contracts, proposes a property-guided method to compute contracts with the help of Horn solvers, and provides experimental evidence that such an approach indeed works on a selection of SV-COMP benchmarks. Our implementation, at this point, is a proof of concept, and more work is needed to handle real-world C programs: in particular, inference not only of contracts but also of loop invariants, and inference of contracts also in the presence of arrays, heap-allocated data-structures, and pointers. Within those restrictions, we believe that the experimental results are encouraging, and that the proposed combination of deductive verification technology with model checking algorithms can significantly extend the reach of both paradigms.

Acknowledgements. This work has been partially funded by the Swedish Governmental Agency for Innovation Systems (VINNOVA) under the AVerT project 2018-02727, by the Swedish Research Council (VR) under grant 2018-04727, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 789–801. ACM (2016). https://doi.org/10.1145/2837614.2837628
3. Alshnakat, A.: Automatic verification of embedded systems using horn clause solvers. Master's thesis, Uppsala University, Department of Information Technology (2019)
4. Benveniste, A., et al.: Contracts for system design. Found. Trends Electron. Des. Autom. **12**(2–3), 124–400 (2018)
5. Beyer, D.: Software verification and verifiable witnesses. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 401–416. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_31
6. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
7. Claessen, K., Smallbone, N., Hughes, J.: QuickSpec: guessing formal specifications using testing. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 6–21. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13977-2_3
8. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
9. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
10. Danial, A.: Cloc - count lines of code. http://cloc.sourceforge.net/
11. Denney, E., Fischer, B.: A generic annotation inference algorithm for the safety certification of automatically generated code. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (eds.) Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, 22–26 October 2006, Proceedings, pp. 121–130. ACM (2006), https://doi.org/10.1145/1173706.1173725
12. Dijkstra, E.W.: Guarded commands, non determinacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). http://doi.acm.org/10.1145/360933.360975
13. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976). http://www.worldcat.org/oclc/01958445

14. Ernst, M.D., et al.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015
15. Gordon, M., Collavizza, H.: Forward with Hoare. In: Roscoe, A.W., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C.A.R. Hoare. LNCS, pp. 101–121. Springer, London (2010). https://doi.org/10.1007/978-1-84882-912-1_5
16. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI, pp. 405–416 (2012). http://doi.acm.org/10.1145/2254064.2254112
17. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
18. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). http://doi.acm.org/10.1145/363235.363259
19. Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, 30 October–2 November 2018. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603013
20. Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn clauses for communicating timed systems. In: Bjørner, N., Fioravanti, F., Rybalchenko, A., Senni, V. (eds.) Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014. EPTCS, vol. 169, pp. 39–52 (2014). https://doi.org/10.4204/EPTCS.169.6
21. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
22. Jones, C.B.: Developing methods for computer programs including a notion of interference. Ph.D. thesis, University of Oxford, UK (1981). http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.259064
23. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: a framework for verifying java programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 352–358. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_19
24. Knüppel, A., Thüm, T., Padylla, C., Schaefer, I.: Scalability of deductive verification depends on method call treatment. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 159–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_15
25. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
26. Meyer, B.: Applying "design by contract". IEEE Comput. **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
27. Moy, Y.: Sufficient preconditions for modular assertion checking. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 188–202. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78163-9_18
28. Nyberg, M., Gurov, D., Lidström, C., Rasmusson, A., Westman, J.: Formal verification in automotive industry: enablers and obstacles. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 139–158. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_14
29. Oheimb, D.: Hoare logic for mutual recursion and local variables. In: Rangan, C.P., Raman, V., Ramanujam, R. (eds.) FSTTCS 1999. LNCS, vol. 1738, pp. 168–180. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-46691-6_13

30. Owe, O., Ramezanifarkhani, T., Fazeldehkordi, E.: Hoare-style reasoning from multiple contracts. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 263–278. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_17
31. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. **6**, 319–340 (1976). https://doi.org/10.1007/BF00268134
32. Polikarpova, N., Ciupa, I., Meyer, B.: A comparative study of programmer-written and automatically inferred contracts. In: Rothermel, G., Dillon, L.K. (eds.) Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, 19–23 July 2009, pp. 93–104. ACM (2009). https://doi.org/10.1145/1572272.1572284
33. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_20
34. Seghir, M.N., Kroening, D.: Counterexample-guided precondition inference. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 451–471. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_25
35. Seghir, M.N., Schrammel, P.: Necessary and sufficient preconditions via eager abstraction. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 236–254. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12736-1_13
36. Singleton, J.L., Leavens, G.T., Rajan, H., Cok, D.R.: Inferring concise specifications of APIs. CoRR abs/1905.06847 (2019). http://arxiv.org/abs/1905.06847
37. SV-Comp: Collection of verification tasks. https://github.com/sosy-lab/sv-benchmarks