Requirements for Constraint Solvers in Verification of Data-Intensive Embedded System Software*

Qiang Fu[1], Maurice Bruynooghe[1], Gerda Janssens[1], and Francky Catthoor[2]

[1] Katholieke Universiteit Leuven, Department of Computer Science,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
qiang,maurice,gerda@cs.kuleuven.be
[2] IMEC vzw, Kapeldreef 75, B-3001 Heverlee, Belgium
catthoor@imec.be

Abstract. In tuning data-intensive software such as multimedia and telecom applications for embedded processors in portable devices, designers use a combination of automated and manual transformations at the source level to optimize the resource consumption of the software. It is of crucial importance that the functionality of the software is preserved. For software with static control, a verification method exists that first transforms the code into dynamic single assignment form and next verifies the functional equivalence of the two versions. The verification is based on geometric modelling using polyhedra.
In this paper, we describe in detail the basic operations of the verification method, discuss the control issues that affect its overall performance, and analyze the functionalities that constraint solvers have to offer to handle this application.

1 Introduction

Embedded systems for the consumer electronics market run data-intensive multimedia and telecom applications on devices with severe resource constraints. Designing such systems is a complex task. Typically, designers start from an initial design assembled by a straightforward combination of trusted algorithms in a high level language. Then designers want to optimize performance, area on chip, power consumption and overall cost of the design. For that purpose, they use analysis tools and perform a mix of automated and manual transformations according to some design methodology (e.g. [4]) to parallelize the loops or to improve the array related memory management. Figure 1 illustrates a typical loop transformation which would benefit the memory size. The whole process is very error prone. Moreover, once the design in the high level language is frozen and implemented in the embedded system, the cost of bugs in the design becomes excessive. Hence there is a need for thorough testing and/or verification.

* Work supported by FWO-Vlaanderen

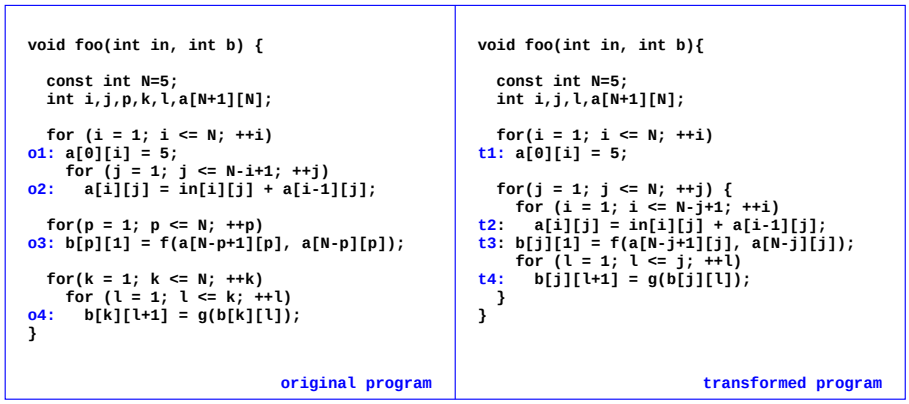

Fig. 1. Program before and after loop transformation.

For (parts of) systems with static control flow where conditions, index expressions, and bounds on iterators are (piecewise) affine functions of the bounds of the surrounding iterators and where no pointer references occur, the code can be transformed to so called Dynamic Single Assignment (DSA) code [5] where each array element is written only once by methods described in [5,15]. This meets the requirements of a very relevant subset of all possible application codes in our target domain. The resulting code is *functional*: Each element of an output array is a function of a set of elements of the input arrays. Verification of the equivalence of original and transformed code then reduces to checking for each output element that the function mapping inputs to outputs is equivalent. In order to ensure scalability to realistic data and loop sizes in the embedded system domain, the crux of the methods is to do this verification not element by element but to handle at once groups of elements for which the function is the same. Such methods are described in [1,13].

Those methods rely on the use of the *geometric model* (also called polyhedral/polytope model) for representing the meaning of programs. Geometric modelling of programs is well known in the parallel compiler and regular array synthesis research domains and is used extensively to analyze the execution of program statements [5,11,12]. The geometric model concisely represents all of the necessary information about the data and control flow in the program. The basic idea is to represent the iterations for which a statement is executed by the integer points in a polytope, i.e., a bounded polyhedron. A polyhedron is a subspace in n-dimensional space bounded by a finite number of hyperplanes. These hyperplanes can be represented as a system of linear inequalities. The latter can be extracted from the iterator bounds and conditions that control the execution of the statement.

This paper develops in more detail the method sketched in [13], discusses control issues that affect the performance, and analyzes the functional requirements for the constraint solving. In Section 2, we explain the geometric modelling of the

program. The concept of proof obligations is introduced in Section 3. Also the basic operations to reduce proof obligations together with the functionality they require from the underlying constraint solvers are described in this section. In Section 4, we discuss how to avoid redundant computations during the reduction of proof obligations. Handling recurrences is presented in Section 5. In Section 6, we discuss in more detail which operations are required and how existing solvers provide support for them. Section 7 concludes.

2 Program representation under the geometric model

Geometric modelling is used by many authors. Here we recall the basics by means of some examples. Consider statement *o2* in Figure 1 (with `N` substituted by its value) which is a **writer** of array `a` and a **reader** of arrays `in` and `a`:

```
    for (i = 1; i <= 5; ++i)
       for (j = 1; j <= 5-i+1; ++j)
o2:        a[i][j] = in[i][j] + a[i-1][j];
```

The **iteration domain** is a relation over the iterators governing the statement. Each tuple (i, j) in the relation defines a value of the iterators for which the statement is executed. The relation is described by the integer points in a polytope: $D = \{(i, j) \mid i \geq 1 \wedge i \leq 5 \wedge j \geq 1 \wedge j \leq 5 - i + 1\}$.

The **definition domain** is a relation over the indices of the array in the left hand side of the statement. It defines which elements of the array are written by the statement. Also this relation can be described by the integer points in a polytope: $W_{\mathtt{a}} = \{(a_1, a_2) \mid \exists i, j : a_1 = i \wedge a_2 = j \wedge (i, j) \in D\} = \{(a_1, a_2) \mid \exists i, j : a_1 = i \wedge a_2 = j \wedge i \geq 1 \wedge i \leq 5 \wedge j \geq 1 \wedge j \leq 5 - i + 1\}$. Because the program is in DSA form, the constraints define a bijection between the indices a_1 and a_2 of the array and the iterators i and j. Using the two equalities, the existentially quantified variable i and j can be eliminated and one obtains $W_{\mathtt{a}} = \{(a_1, a_2) \mid a_1 \geq 1 \wedge a_1 \leq 5 \wedge a_2 \geq 1 \wedge a_2 \leq 5 - a_1 + 1\}$.

For each operand in the right hand side, one can define an **operand domain**. It is a relation over the indices of the operand array. It defines which elements of the array are read by the statement. Similar to the definition domain, it can be described by the integer points in a polytope. For the second operand, it is given by: $R_{\mathtt{a}} = \{(a_1, a_2) \mid \exists i, j : a_1 = i - 1 \wedge a_2 = j \wedge (i, j) \in D\}$. Note that there is a functional dependency from the iterators to the indices as one value is read in each iteration, but in general not from the indices to the iterators as the same value can be read in different iterations. Again, i and j can be eliminated and we obtain $R_{\mathtt{a}} = \{(a_1, a_2) \mid a_1 + 1 \geq 1 \wedge a_1 + 1 \leq 5 \wedge a_2 \geq 1 \wedge a_2 \leq 5 - a_1\}$.

Each executed instance of the statement reads values from elements in the operand arrays and writes a value in an element of the lhs array. For each operand, there is a **dependence mapping** that defines which operand element is read for each written element. As said above, the relation between the indices of the lhs array and the iterators is a bijection while there is functional dependency between the iterators and the indices of the operand array, hence the dependency

mapping can be understood as a function from the indices of the lhs array to the indices of the operand array (and is in general not invertible). To stress that the dependence mapping encodes a functional dependency, we denote it as $M(\boldsymbol{i} \rightarrow \boldsymbol{j})$ with $\boldsymbol{i}$ the indices of the written array and $\boldsymbol{j}$ the indices of the operand array. Also this relation can be represented by the integer points of a polytope. For the second operand of our example, we have: $M((a'_1, a'_2) \rightarrow (a_1, a_2)) = \{a'_1, a'_2, a_1, a_2 \mid \exists i, j : a'_1 = i \wedge a'_2 = j \wedge a_1 = i - 1 \wedge a_2 = j \wedge (i, j) \in D\} = \{a'_1, a'_2, a_1, a_2 \mid a'_1 = a_1 + 1 \wedge a_2 = a'_2 \wedge a'_1 \geq 1 \wedge a'_1 \leq 5 \wedge a'_2 \geq 1 \wedge a'_2 \leq 5 - a'_1 + 1\}$. In general, with m the dimension of the array being written and n the dimension of the array being read, the dependence mapping is a polytope in a space of dimension $m + n$. The dimension of the polytope however can be lower, because the constraints can imply equalities.

Statements can be written in a **normal form** where all indices of the lhs array are distinct variables and the indices of the rhs arrays are functions of the indices of the lhs side. The latter correspond to the dependency mapping of the statement. This normal form, together with the constraints on the indices forms the *geometric model of the statement* and completely characterizes it. The statement *o2* is already in normal form as the indices are the iterators. The constraints are those of the iterator domain.

While in the above examples, the relation of interest is represented by *all* integer points inside a polytope, this is not always the case. Consider for example a for loop with a non-unit stride:

```
for i=1, i<=21; i= i+5
```

Its iteration domain is modelled by the formula $D = \{(i) \mid \exists k : i \geq 1 \wedge i \leq 21 \wedge i = 1 + 5k\}$ which contains an existentially quantified variable. The relation represents the points in the set $\{1, 6, 11, 16, 21\}$; these are not all the integer points inside $(i \geq 1 \wedge i \leq 21)$ which is the projection of the original two dimensional polyhedron $(i \geq 1 \wedge i \leq 21 \wedge i = 1 + 5k)$ upon i. The existentially quantified variable is also introduced when normalizing a statement such as `a[2i] = ...`. Indeed, normalization will replace it with `a[k] = ...` and compute a constraint $(\exists i : k = 2i \wedge \ldots \leq i \leq \ldots)$ in a normal form. Also modulo operations and integer division can give rise to such variables. These formulae with existentially quantified variables belong to the class of the Presburger formulae.

Now, a program can simply be represented by the geometric models of the statements. The dependencies between reads and writes are captured by the use-def chains which can be derived from the geometric models of the statements. Note that the exact execution order is not modelled since it is irrelevant to the verification purpose.

3 Verification method

Our verification task consists of proving that the original and the transformed programs compute the same outputs when their inputs are equal. To do so, one is given the names of the corresponding input and output arrays. Also, one can

assume that the functions called by programs are side-effect free and have not been modified by the transformation, that the programs are in DSA form and that the data flow is correct, i.e., that values are read after being written (or are available as input). In what follows, we use some notational conventions. The arrays in the original and the transformed program are distinguished by the respective superscript $^\circ$ and $^\mathsf{t}$. A vector $(i_1, \ldots, i_n)$ is denoted as $\boldsymbol{i}$; i_l refers its l^{th} element. In the verification task for the programs of Figure 1, the corresponding input arrays are $\mathtt{in}^\circ$ and $\mathtt{in}^\mathsf{t}$; the corresponding output arrays are $\mathtt{b}^\circ$ and $\mathtt{b}^\mathsf{t}$.

The verification task can be expressed by a (conjunction of) proof obligation(s). A **proof obligation** describes an equivalence relation that must hold between one expression from the original program and the other one from the transformed program. It is formalized as a tuple $(exp^\circ(\boldsymbol{i}), exp^\mathsf{t}(\boldsymbol{j}), P(\boldsymbol{i}, \boldsymbol{j}))$, where $exp^\circ(\boldsymbol{i})$ and $exp^\mathsf{t}(\boldsymbol{j})$ are expressions from the original and transformed program respectively; these expressions are parameterized by respective vectors of variables $\boldsymbol{i}$ and $\boldsymbol{j}$. $P(\boldsymbol{i}, \boldsymbol{j})$ is a set of constraints that specifies a relation between the vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ (the tuples in the relation are the integer points in the polytope). The meaning is: for each pair $(\boldsymbol{i}, \boldsymbol{j}) \in P(\boldsymbol{i}, \boldsymbol{j})$ (i.e., all integer solutions of P) the equality $exp^\circ(\boldsymbol{i}) = exp^\mathsf{t}(\boldsymbol{j})$ has to be proven.

For example, the verification task of Figure 1 can be modelled by the proof obligation $(\mathtt{b}^\circ[i_1, i_2], \mathtt{b}^\mathsf{t}[j_1, j_2], P((i_1, i_2), (j_1, j_2)))$, in which $P((i_1, i_2), (j_1, j_2)) = \{(i_1, i_2), (j_1, j_2) \mid i_1 = j_1 \wedge i_2 = j_2 \wedge i_1 \geq 1 \wedge i_1 \leq 5 \wedge i_2 \geq 1 \wedge i_2 \leq i_1 + 1 \wedge j_1 \geq 1 \wedge j_1 \leq 5 \wedge j_2 \geq 1 \wedge j_2 \leq j_1 + 1\}$. This proof obligation between output arrays $\mathtt{b}^\circ$ and $\mathtt{b}^\mathsf{t}$ expresses that both programs are equivalent if one proves that $\mathtt{b}^\circ[i_1, i_2] = \mathtt{b}^\mathsf{t}[j_1, j_2]$ for all pairs $(\boldsymbol{i}, \boldsymbol{j}) \in P$. While in the initial proof obligation, the relation between $\boldsymbol{i}$ and $\boldsymbol{j}$ is a bijection, this is in general not the case. Also, the given correspondence between input arrays can be expressed in this form (as assumptions). For our example it can be formulated as : $(\mathtt{in}^\circ[i_1, i_2], \mathtt{in}^\mathsf{t}[j_1, j_2], \{(i_1, i_2), (j_1, j_2) | i_1 = j_1 \wedge i_2 = j_2 \wedge i_1 \geq 1 \wedge i_1 \leq 5 \wedge i_2 \geq 1 \wedge i_2 \leq 5)\}$.

The verification method then consists of using the geometric models of the program statements to reduce the initial proof obligations to proof obligations that trivially hold because they belong to the assumptions.

Reduction of proof obligations There are three basic reduction steps:

Reduction I Peeling expressions in a proof obligation of the form

$$(f(exp_1^\circ(\boldsymbol{i}), \ldots, exp_n^\circ(\boldsymbol{i})), f(exp_1^\mathsf{t}(\boldsymbol{j}), \ldots, exp_n^\mathsf{t}(\boldsymbol{j})), P(\boldsymbol{i}, \boldsymbol{j}))$$

The method leaves the functions uninterpreted and imposes (as sufficient condition) that the arguments of the functions must be pairwise equivalent[3]. Hence the proof obligation is replaced by the conjunction of n proof obligations of the form $(exp_k^\circ(\boldsymbol{i}), exp_k^\mathsf{t}(\boldsymbol{j}), P(\boldsymbol{i}, \boldsymbol{j}))$. The proof fails if different top level function symbols are found. This indicates an error in the transformation.

[3] See [13] for a discussion how to extend the approach for handling commutative and associative functions.

Reduction II Propagation across an assignment. If one of the expressions is an array reference, the corresponding writers of the array can be used to reduce the proof obligation. Without loss of generality, let us assume a proof obligation of the form $(\mathtt{a}^{\circ}[\boldsymbol{f(i)}], exp^{\mathtt{t}}(\boldsymbol{j}), P(\boldsymbol{i},\boldsymbol{j}))$, i.e., with an array reference from the original program. Let the normalized statement s: $\mathtt{a}^{\circ}[\boldsymbol{k}] = exp^{\circ}(\boldsymbol{k})$ be a writer of $\mathtt{a}^{\circ}$ and $C(\boldsymbol{k})$ the constraints on $\boldsymbol{k}$. Using the set of equalities $\boldsymbol{k} = \boldsymbol{f(i)}$, the proof obligation can be rewritten as $(\, exp^{\circ}(\boldsymbol{f(i)}), exp^{\mathtt{t}}(\boldsymbol{j}), P(\boldsymbol{i},\boldsymbol{j}) \wedge C(\boldsymbol{f(i)})\,)$ with $C(\boldsymbol{f(i)})$ the constraint with the elements of $\boldsymbol{k}$ substituted by the corresponding elements from $\boldsymbol{f(i)}$.
The new proof obligation can be discarded when the constraint is inconsistent, i.e., when none of the values referred to by $\mathtt{a}^{\circ}[\boldsymbol{f(i)}]$ is actually written by s. Doing the propagation for each of the writers of $\mathtt{a}^{\circ}$, the original proof obligation is replaced by one proof obligation for each of the writers that actually writes some of the values referred to by $\mathtt{a}^{\circ}[\boldsymbol{i}]$.

Reduction III When both expressions are references to input arrays, i.e., of the form $(\mathtt{in}^{\circ}[\boldsymbol{f(i)}], \mathtt{in}^{\mathtt{t}}[\boldsymbol{g(j)}], P(\boldsymbol{i},\boldsymbol{j}))$, the proof obligation can be dismissed as satisfied after checking that it can be proved from the given assumptions about the correspondence between input arrays. More precisely, given the input equivalence assumption $(\mathtt{in}^{\circ}[\boldsymbol{k}], \mathtt{in}^{\mathtt{t}}[\boldsymbol{l}], C(\boldsymbol{k},\boldsymbol{l}))$, the integer points in $P(\boldsymbol{i},\boldsymbol{j})$ are a subset of those in $C(\boldsymbol{k},\boldsymbol{l})$ under the condition: $\boldsymbol{k} = \boldsymbol{f(i)} \wedge \boldsymbol{l} = \boldsymbol{g(j)}$, i.e., that $P(\boldsymbol{i},\boldsymbol{j}) \wedge \neg C(\boldsymbol{k},\boldsymbol{l})$ is unsatisfiable (or that $C(\boldsymbol{k},\boldsymbol{l})$ is entailed by $P(\boldsymbol{i},\boldsymbol{j})$).

The reduction of proof obligations requires the following primitive operations on relations (constraints): emptiness checking (consistency checking) and subset testing (entailment or negation).

However, a naive application of the above method results in a rather inefficient procedure because: ① Several statements can read (the same or different) values written by some statements, hence several proof obligations involving the same statement can be created. ② Recurrences (direct or indirect) are processed by complete loop unrolling, which is definitely not feasible in practice.

4 Control issues in the absence of recurrences

A sequence of different statements $s_0, s_1, \ldots, s_{n-1}$ represents a **recurrence** when each statement s_i is a writer of an array $\mathtt{a_i}[\boldsymbol{k_i}]$ and a reader of an array $\mathtt{a_{i+1}}[\boldsymbol{l_i}]$ with dependency mapping $M(\boldsymbol{k_i} \rightarrow \boldsymbol{l_i})$, $\mathtt{a_0}$ and $\mathtt{a_n}$ are the same array, and the equijoin of dependency mappings[4] i.e., $M(\boldsymbol{k_0} \rightarrow \boldsymbol{k_n}) = M(\boldsymbol{k_0} \rightarrow \boldsymbol{k_1}) \wedge M(\boldsymbol{k_1} \rightarrow \boldsymbol{k_2}) \wedge \ldots \wedge M(\boldsymbol{k_{n-1}} \rightarrow \boldsymbol{k_n})$ is a nonempty relation. $M(\boldsymbol{k_0} \rightarrow \boldsymbol{k_n})$ is called a *self dependence mapping about array* $\mathtt{a_0}$. Note that a recurrence implies that each statement s_i is a writer of a value used to compute another value it writes; in particular, for statement s_0, we have that $\mathtt{a_0}[\boldsymbol{k_n}]$ is used to compute $\mathtt{a_0}[\boldsymbol{k_0}]$ for each tuple $(\boldsymbol{k_0}, \boldsymbol{k_n})$ in M.

[4] The dependency mapping can be viewed both as a relation and as a constraint, under the constraint view, the equijoin can be denoted as a conjunction of constraints.

In this section we discuss how to tackle the inefficiencies caused by multiple reads from elements written by the same statement in the absence of recurrences.

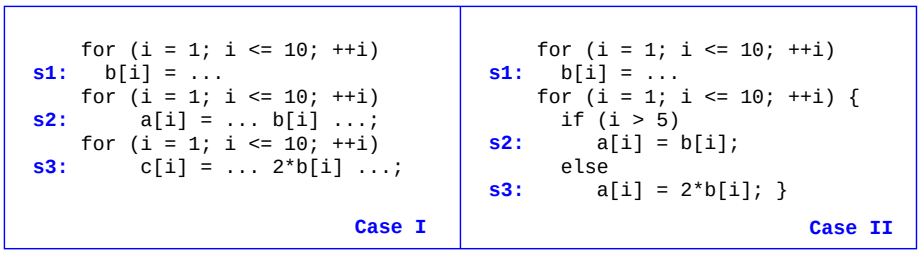

Fig. 2. Two programs illustrating the need for a good control.

Case I of Figure 2 shows a program where array `b` written in statement *s1* is read two times. This will give rise to two different proof obligations involving `b`$[i]$ with the same boundaries on i. One should avoid proving it twice (there can be many steps before the input is reached).

Case II of the same figure shows a slightly different circumstance. Now, the two proof obligations refer to different pieces of the array `b`, so they will be different. However, substantial work could be saved if one could merge both proof obligations into a single one. Indeed, then only one proof obligation need be reduced to a condition between input arrays instead of two.

A simple way to tackle the inefficiency of Case I is by tabling all proof obligations. A new proof obligation can be dismissed if it is implied by an already tabled one (in the same way as a proof obligation between input arrays can be dismissed when implied by the assumptions, see Section 3). Permanently storing all proof obligations that occur during the verification may result in the huge table size.

In the absence of recurrences, it is possible to define a strategy that avoids such redundancies and keeps table size to a minimum by storing only the necessary proof obligations. Our strategy is to associate a counter with each statement s; this counter is initialized with the number of readers of the array written by s. Now consider a proof obligation of the form $(\mathtt{a}[\boldsymbol{f(i)}], exp^{\mathtt{t}}(\boldsymbol{j}), P(\boldsymbol{i},\boldsymbol{j}))$ and let *s0* be one of the writers of `a`. The Reduction II step propagating the proof obligation across *s0* is delayed until the counter associated with statement *s0* is 0, i.e., until all proof obligations originating from readers of elements of `a` that are written by *s0* are available (and redundant ones have been removed). After propagation, all counters of writers of arrays $\mathtt{b_i}$, for which *s0* is a reader, can be decreased by 1 as all proof obligations originating from the reader *s0* are now available. This works fine when *s0* is a copy statement as the new proof obligation is ready for propagation. However, when the rhs is an expression, then the proof obligation about $\mathtt{b_i}$ is not ready for propagation until one or more peel steps have been

applied. It blocks propagation across the writers of $\mathtt{b}_i$ until it has been further reduced by peel steps.

Our strategy implies that the propagation steps across statements are bundled in a different way. Instead of propagating one proof obligation $(\mathtt{a}[i], \ldots)$ across all writers of `a` at once, all proof obligations containing `a` are at once propagated across a single writer of `a` (when its counter is 0 and there are no pending peel operations). The correctness of the dataflow together with the absence of recurrences ensures that the verification will never be blocked as there will always be a zero counter.

A proof obligation with an array reference is tabled when created and remain *active* until it has been propagated to all writers of that array, at which point it can be removed. Hence the table consists of a set of active proof obligations which is only a fraction of the total number of proof obligations that are created during the verification.

5 Handling recurrences

Figure 1 contains two recurrences consisting of a single statement (*direct recurrences*), namely statements *o2* and *o4*. Also, there exists *indirect recurrences* in which several statements are involved. Figure 3 shows another example with a direct recurrence in statement *s3*. Note that the program slice that computes the value of $a[7]$ also contains the instances of *s3* for $i = 5$ and $i = 3$.

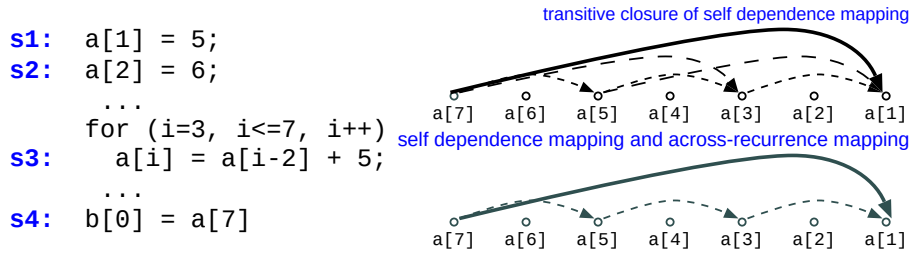

Fig. 3. A piece of code with a recurrence.

The control described in Section 4 will be blocked at recurrences. Consider the example; the counter of *s3* will never reach 0 because one of the readers of the values it writes is inside the recurrence. Simply not counting the reads inside the recurrence when initializing the counters will avoid the deadlock but will result in a naive algorithm that unrolls the recurrence, what results in an unacceptable performance degradation. In our example, the recurrence is entered when propagating a proof obligation about `a`[7] reaches *s3*. This is then reduced to a proof obligation about `a`[5] and next about `a`[3]. Finally, the recurrence is exited by the proof obligation about `a`[1].

Note that the dependencies in a recurrence are always well-founded because the program is in DSA and the data flow is correct. Hence there are always statements that exit the recurrence when tracing the dependencies (statements *s2* and *s1* in our example). As we mentioned at the beginning of Section 4, a recurrence can be characterized by a self dependency mapping $M(\boldsymbol{k_0} \rightarrow \boldsymbol{k_n})$ for some array a. If the distance between $\boldsymbol{k_0}$ and $\boldsymbol{k_n}$ is the same for all tuples $(\boldsymbol{k_0}, \boldsymbol{k_n})$ in M then we can use an approach that avoids completely unrolling the recurrence (for other recurrences, we simply use the naive approach mentioned above).

One way, sketched in [13] is to compute the positive transitive closure of the self dependence mapping (the arrows in the top right of Figure 3) and to use the difference between domain and range of the transitive closure relation to compute the across-recurrence mapping. Another approach is to combine the constant distance of the self-dependency mapping with domain information to extend the relation containing one tuple of the self dependency mapping into a relation covering all tuples of the mapping (the arrows in the bottom right of Figure 3) and to extend the proof obligation in the same way. Note that one must have a recurrence in both the original and the transformed program and that both have to be synchronous. This makes the whole technique rather involved and we omit further details.

6 Solvers

6.1 Requirements

As we have seen in Section 2, for simple statements, the relations of interest can be represented by the integer points in a polytope. However, for more complex statements, more complex constraints are required that involve existentially quantified variables. These existentially quantified variables cannot be eliminated by projection because, as we illustrated, the set of integer points in the projection of a polyhedron can be strictly larger than the set obtained by projecting the integer points in the original polyhedron. In other words, projection can introduce an overestimation; as our verification method requires exact modelling, it is not a safe operation.

As a summary, the basic operations required by the verification method are consistency checking (does the constraint has an integer solution) and entailment. Another useful operation is convex hull. It can be used to merge several active proof obligations into a single one, and hence to reduce the size of the table (Section 4) and the number of reduction steps (Section 3). In fact, there are two ways to simplify the set of active proof obligations between the same pair of expressions:

- When the constraint part of one proof obligation is entailed by the constraint part of another one, it can be discarded.
- When the convex hull of the constraints of two proof obligations entails their disjunction (in other words, the convex hull is equivalent to the disjunction),

they can be replaced by a single proof obligation that has the convex hull as constraint.

The example below (taken from case II in fig 2) shows an application of this simplification.

Example 1. In the program, statement *s2* gives rise to a proof obligation of the form $(\mathtt{a}[i], exp(\boldsymbol{k}), i \geq 1 \wedge i \leq 5 \wedge C(\boldsymbol{k}))$. A similar proof obligation $(\mathtt{a}[j], exp(\boldsymbol{l}), j \geq 6 \wedge j \leq 10 \wedge C(\boldsymbol{l}))$ is raised by statement *s3*. Imposing the equalities $i = j$ and $\boldsymbol{k} = \boldsymbol{l}$ one can derive that the convex hull is given by $(i \geq 1 \wedge i \leq 10 \wedge C(\boldsymbol{k}))$ and that it is equivalent to the disjunction $(i \geq 1 \wedge i \leq 5 \wedge C(\boldsymbol{k})) \vee (i \geq 6 \wedge i \leq 10 \wedge C(\boldsymbol{k}))$, hence both proof obligations can be replaced by $(a[i], exp(\boldsymbol{k}), i \geq 1 \wedge i \leq 10 \wedge C(\boldsymbol{k}))$.

6.2 Solvers

In simple verification tasks, all constraints correspond to polytopes (the constraints do not contain existential variables). However, solutions are the integer points in these polytopes, hence, more is required than basic capabilities for solving linear equalities and inequalities over rationals or reals.

CLP(Q) [3] is a library in SICSTUS Prolog [14] for solving linear programming problems with a limited support for mixed integer linear optimization problem. Besides the consistency check over the rational domain, it allows one to check that there is at least one integer solution (using the `bb_inf` operation). Using these primitive operations, one can build more complex operations needed by our verification. As described in [2], a convex hull operation can be constructed.

The PolyLib [8] library is a software package that is designed for manipulating polyhedra. PolyLib is designed to handle *polyhedral domains* which refer to the set of integer points in the union of a finite number of polyhedra. It provides functions for various operations including testing for the existence of an integer solution and calculating convex hull. So it provides all the functionality for handling simple verification tasks.

PPL [10] is another library, however it is oriented more towards the support of rational convex polyhedra, and includes the ability to handle the strict inequalities. But the lack of support for checking the existence of integer solutions makes it not suitable for our verification task.

When it comes to the verification of more complex problems involving existential variables, then all of the above systems are not suited. For certain kinds of constraints there may be work-arounds that eliminate the existential variables. In particular $\mathcal{Z}$-polyhedra [9] may be useful to represent certain types of constraints with existential variables. Also PIP [6], a tool which computes the lexicographic minimum of the integer points in a parametric polyhedron, can sometimes be helpful in eliminating certain existential variables [17]. However, not all existential variables can be eliminated. In such case we need a solver that supports general Presburger formulae that contains existentially quantified variables.

The Omega library [11] is designed specifically for handling full Presburger formulae. It is based on an extension of the Fourier-Motzkin method called Omega test. It also provides the other operations required by our application, such as entailment, convex hull, and even simplification that replaces a disjunction with its convex hull when they are equivalent. It also provides a transitive closure operation [7].

Presburger formulae have a super-exponential time complexity. Hence Omega necessarily employs various heuristics. Sometimes, these heuristics may fail. Then the calculation either continues running without returning a solution in a reasonable time or simply gives an UNKNOWN result, as reported in Section 5.2.1 of [16]. That is likely inherent to any solver for the general class of Presburger formulae. Moreover, as reported by [16], the implementation of Omega has other problems that may cause it to abort the calculation with an error message, or in very rare cases even produce incorrect results.

7 Conclusion

In this paper we analyzed a verification task for embedded software that was previously sketched in [13]. We described in more detail the basic operations of the verification process. We also analysed the functionality a solver has to offer to be useable in this application. For simple verification tasks, where existential variables do not appear, or can be eliminated by simple work-arounds, various solvers that can handle polyhedra can be applied. However, for more complex verification tasks, only Omega[11] offers all the needed functionality, though there are no guarantees that it will never fail.

Acknowledgement

We are grateful to Peter Vanbroekhoven and Sven Verdoolaege for the many discussions and useful comments.

References

1. D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. Technical Report Report RR-4285, INRIA, Oct. 2001.
2. F. Benoy, A. King, and F. Mesnard. Computing Convex Hulls with a Linear Solver. *Theory and Practice of Logic Programming*, 5:259–271, 2005.
3. H. C. OFAI CLP(Q,R) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.
4. F. Catthoor, S. Wuytack, E. de Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.
5. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

6. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, Sep 1998.
7. W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *International Journal of Parallel Programming*, 24(6), 1996.
8. V. Loechner. PolyLib: A library for manipulating parameterized polyhedra. Technical Report PI-785, IRISA, 1999.
9. S. P. K. Nookala and T. Risset. A Library for Z-polyhedral Operations. Technical Report PI-1330, IRISA, Mai 2000.
10. PPL. http://www.cs.unipr.it/ppl/.
11. W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Comm. of the ACM*, Aug 1992.
12. F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Prog. Lang. Syst.*, 22(5):773–815, Sep 2000.
13. K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *Compiler Construction, 14th International Conference, CC 2005, Proceedings*, volume 3443 of *LNCS*, pages 221–236. Springer, 2005.
14. SICSTUS. http://www.sics.se/isl/sicstus.html.
15. P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. Transformation to dynamic single assignment using a simple data flow analysis. In *Proceedings of The Third Asian Symposium on Programming Languages and Systems, Tsukuba, Japan*, 2005.
16. S. Verdoolaege. *Incremental loop transformations and enumeration of parametric sets*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 2005.
17. S. Verdoolaege, K. Beyls, M. Bruynooghe, and F. Catthoor. Experiences with enumeration of integer projections of parametric polytopes. In *Compiler Construction, 14th International Conference, CC 2005, Proceedings*, volume 3443 of *LNCS*. Springer, 2005.