A JastAdd- and ILP-based Solution to the Software-Selection and Hardware-Mapping-Problem at the TTC 2018

Sebastian Götz, Johannes Mey, Rene Schöne and Uwe Aßmann
{first.last}@tu-dresden.de, sebastian.goetz@acm.org

Software Technology Group
Technische Universität Dresden

Abstract

The TTC 2018 case describes the computation of an optimal mapping from software implementations to hardware components for a given set of user requests as a model transformation problem. In this paper, we show a detailed view on the reference solution which uses two main approaches: 1) transformation using attribute grammars and higher-order attributes into an integer linear programming (ILP) specification, and 2) solving the ILP resulting in a valid and optimal mapping. We further show evaluation results for the given scenarios.

1 Introduction

The TTC 2018 case "Quality-based Software-Selection and Hardware-Mapping as Model Transformation Problem" describes a complex problem. A system is described by its software components with variants, typed hardware resources and requests for certain software components. A variant of a component may require other dependent components, which themselves may have dependencies. The problem is now to a) select a variant for the requested software components and their required components, and b) find a mapping of those components to suitable hardware resources. The overall solution to the variant selection problem is in the form of a directed acyclic graph. Since each usage of a variant must be mapped to a new resource, this results in a tree shaped solution for the combined problem. Such a mapping has to obey the constraints defined for each software variant, i.e., it has to fulfill the "contract". Further, a mapping is optimal, if the value of the specified objective function is optimal, e.g., if the projected energy consumption is minimal. The value of the objective function depends on the selection of a software variant, as well as on the resource it is deployed on. The complexity of the problem arises from the fact, that selection and mapping influence each other, thus can not be solved independently.

The solution is publicly available[1] within the module `jastadd-mquat-solver-ilp`. The rest of the paper describes our solution in detail (Section 2), shows our evaluation results (Section 3) and concludes in Section 4.

2 Solution

The input model is given as a set of Java objects describing nodes in a tree, whose classes are generated by the JastAdd framework [EH07] based on a given grammar. This solution uses the approach of reference attribute grammars (RAGs) [Hed00] specifying computations for certain node types. In this solution, we use ILP as a means to solve both subproblems by enumerating all possible mappings, and thereby selecting a variant. The

In: A. Garcia-Dominguez, G. Hinkel, F. Krikava (eds.): Proceedings of the Transformation Tool Contest 2018, Toulouse, France, 29-Jun-2018, published at http://ceur-ws.org

[1] `https://git-st.inf.tu-dresden.de/stgroup/ttc18`

```
ILP ::= IlpObjective IlpConstraint* IlpBound* IlpVariable* <Info:IlpVarInfo> ;
TimedOutILP:ILP ::= <Reason:String> ;

// objective kind is either minimize or maximize
IlpObjective ::= <Kind:IlpObjectiveKind> IlpLeftHandSide ;

IlpConstraint ::= <Name:String> IlpLeftHandSide <ClauseComparator:ClauseComparator> <RightHandSide:double> ;

IlpBound ::= <Ref:IlpVariable> <Type:IlpBoundType> ;

IlpVariable ::= <Name:String> <Request:Request> <Impl:Implementation> ;
IlpAllResourcesVariable:IlpVariable ;
IlpMappingVariable:IlpVariable ::= <Resource:Resource> ;

// sum of terms
IlpLeftHandSide ::= IlpTerm* ;

IlpTerm ::= <Value:double> <Ref:IlpVariable> ;
```

Figure 1: The target model describing an ILP

$$
\begin{array}{ll}
\text{minimize} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\
\text{subject to} & A\mathbf{x} \leq \mathbf{b}, \\
 & \mathbf{x} \geq \mathbf{0}, \\
\text{and} & \mathbf{x} \in \mathbb{Z}^n,
\end{array}
$$

Figure 2: Canonical form of an integer linear program

remainder of this section describes the transformation problem in detail, first depicting the target model to transform the problem model to (Section 2.1), explain the ideas behind the model structure (Section 2.2), and finally describe the transformation step in Section 2.3.

2.1 The Target Model

The intermediate model is described by the grammar shown in Figure 1. It describes an ILP in its standard form shown in Figure 2. The basic idea is to use binary variables denoting the deployment of a software implementation on a hardware resource for a request. The cross product of all variants, resources and requests yields all possible *deployment variables*. Additionally, there are binary *implementation variables* indicating the usage of a certain implementation for a request independent of its deployment. Those variables are used to create shorter (in-)equations and are usually automatically "inlined" by the solver.

2.2 Details and Concepts in our ILP

As already briefly mentioned in the case description, there are three main kinds of constraints in the ILP. *Architectural constraints* ensure the semantic properties of the problem model:

- For every request, there is exactly one variant of the target component chosen.
- For every component, there is at least one variant chosen. Some component may not be needed in certain configurations.
- On every resource, the can only be at most one deployed component.

Request constraints ensure the constraints set for each request in the problem model. This is done by adding one inequation per constraint with the required property value of the request on the one side and the sum of provided property values of all variants of the target component on the other side. The **ClauseComparator** of the constraint directly determines the kind of its inequation, e.g., the comparator LT implies $\leq$. With this construction, all variants not providing this property are automatically discarded.

The last category, *negotiation constraints*, are the most important, as they ensure the fulfillment of all contracts. This includes:

```
1  for (Clause reqClause : impl.requirementClauses()) {
2    Designator designator = reqClause.getDesignator();
3    IlpLeftHandSide reqLhs = new IlpLeftHandSide();
4    if (designator.isSoftwareDesignator()) {
5      for (Tuple<Implementation, Clause> tuple : reqClause.providingClausesOfRequiredComponent()) {
6        Implementation providingImpl = tuple.getKey();
7        Clause providingClause = tuple.getValue();
8        for (Resource resource : this.getHardwareModel().getResourceList()) {
9          reqLhs.addIlpTerm(new IlpTerm(providingClause.evalUsing(request, resource),
10             getIlpVariable(request, providingImpl, resource)));
11       }
12     }
13     for (Resource resource : this.getHardwareModel().getResourceList()) {
14       // we always use negative eval-value to get the required value on the right side (literally)
15       reqLhs.addIlpTerm(new IlpTerm(makeNegative(reqClause.evalUsing(request, resource)),
16           getIlpVariable(request, impl, resource)));
17     }
18     result.addIlpConstraint(new IlpConstraint(reqLhs, reqClause.getClauseComparator(), 0));
19   } else {
20     for (Resource resource : this.getHardwareModel().getResourceList()) {
21       // check if constraint is fulfilled, otherwise remember this illegal combination
22       if (!reqClause.checkUsing(request, resource)) { setIllegal(request, impl, resource); }
23     }
24   }
25 }
```

Figure 3: Negotiation constraint computation

- If an implementation I requires another component C, an inequation is added of the form: $\sum_{reqI \in C} reqI \geq I$, where I and $reqI$ are implementation variables and $reqI$ are implementations of component C.

- For every **ComponentRequirement** of an implementation on a resource, the following inequation is added: $\sum_{reqClause \in impl} \sum_{provClause, provImpl \in provClauseReqComp(reqClause)} \sum_{resource} provClause * deployVar(provImpl, resource) \leq \sum_{resource} reqClause * deployVar(impl, resource)$, where $provClauseReqComp$ returns all clauses (along with implementations they are defined in) which provide the property of the given clause.

Within the computing attribute, the computation of the negotiation constraint is depicted in Figure 3.

2.3 Transformation to the Target Model and the Solution

To transform the given problem model into the (intermediate) ILP model, we utilize the concept of attribute grammars using the JastAdd tool. Given a grammar and a set of attribute definitions, JastAdd produces executable Java code. Each nonterminal of the grammar is represented by a Java class, within which attributes and model navigation code is generated as methods. Using JastAdd, attributes are methods which can be invoked as usual, e.g., in line 5 of Figure 3. We defined 72 attributes ranging from simple ones, such as navigation inside the model or printing parts of the model, to more complex ones, like the evaluation of clauses or the transformation of the source model to the ILP model. The main motivation for using attributes are the ability to cache their computed values. With that, clauses need only be evaluated once to get their real value for a certain configuration. Another motivation not related to this TTC problem is the possibility to dynamically track dependencies of computed attribute values to enable incremental evaluation if the problem model changes.

Once the intermediate ILP model is generated, we have two possible ways to get a solution for the ILP. The first way is to transform the model to text, i.e., print it to file, and invoke an external solver providing it with the file and let it write its solution to another file. Here, the overhead of reading and writing files arises, which can be significant for larger models. We use GLPK[2] as external solver.

The second way exploits a JNI binding[3] for GLPK, the solver we use. Hence, no file needs to be written, but instead an API is used to construct the internal data structure from the intermediate model. However, this construction is straightforward[4] because the model already resembles this structure.

For both ways, the returned solution needs to be interpreted to construct the solution, i.e., the assignments. This is done by only considering the deployment variables with the value 1 and create an assignment for each

[2]GLPK is available at `https://www.gnu.org/software/glpk/`

[3]The Java-Binding for GLPK is available at `http://glpk-java.sourceforge.net/`

[4]The whole solve method (`ILPDirectSolver.solve0`) of the direct solver needs about 120 lines of code without the logging aspect.

of them. In case of the external solver, this is still possible, because the variable name encodes the names of request, implementation and resource.

2.4 Example Model, ILP and its Solution

To illustrate the process, consider the model defined in Figure 4. There are 3 resources, one main component targeted by the request and requiring two other components. All components have 2 implementations each.

The external ILP solver takes up this model and transforms it to the ILP shown in Figure 5. One might expect to see more variables in the objective function (line 2). However, during the transformation, illegal assignments are already discarded, and, thus not show up in the ILP. Further, the constraints can be seen: Lines 5-9 ensure at most one implementation per component, lines 10-12 ensure define the implementation variables, lines 13-18 are responsible for required components and their properties. Further, lines 19 and 20 represent the request and its constraint, and lines 21-23 ensure at most one deployment per resource.

In this case, the solver finishes successfully, finding an optimal solution. The following variables are found to have the value 1:

- `request0#implementation_0i1`
- `request0#implementation_0i1#resource0`
- `request0#implementation_0i1_0i1`
- `request0#implementation_0i1_0i1#resource1`
- `request0#implementation_0i1_1i0`
- `request0#implementation_0i1_1i0#resource2`

After reconstruction, the assignments shown in Figure 6 result.

3 Evaluation

To investigate the feasibility fo the approach, the implementation was tested with the provided scenarios. The measurements were performed on a rather old and slow Intel i5-3350P machine with 16 gigabytes of memory using Ubuntu 16.04. GLPK 4.65 was using utilizing the GNU MP bignum library. A maximum solving time of 15 minutes per run was allowed; each scenario was executed five times with the two provided solver variants. Table 1 shows the median results of the runs. The tests results are only shown for the small and medium scenario sizes, since on the given hardware, it was not possible to find valid solutions within the given time frame. The table contains information on if a soltuion has been found and if this solution is optimal. Additionally, time measurements for the two solution steps are given. The *generation time* specifies the duration of the generation of the ILP including the GLPK API calls in the *direct* case and the time to write the serialized ILP to disk in the *external* case. The *solving time* specifies the time GLPK requires to solve the model. If a step takes longer than the given maximum time of 15 minutes, *timeout* is stated instead of a duration.

Table 1 shows, that the approach is capable of finding optimal solutions for small problems very quickly. One the other hand, if the problem size and complexity increases, finding valid and, in particular, optimal solutions is very hard.

The acquired measurements allow some further observations with regard to the two presented variants:

- When comparing the direct solver which creates the ILP programmatically to the one which writes the ILP into a file and then calls the external solver, the former variant is always faster. This is mostly caused by the overhead of serializing and deserializing the ILP.
- For larger problems, the generation time is significantly lower than the solving time. Thus, a better optimization or choice of ILP solver may lead to better results.

```
resource resource0:ComputeNode {
 resource cpu0_0:CPU { frequency = 1562.0 load = 0.0 }
 resource ram0:RAM { total = 11494.0 free = 11494.0 }
 resource disk0:DISK { total = 14107.0 free = 14107.0 }
 resource network0:NETWORK { latency = 566.0 throughput = 70960.0
 }
}
resource resource1:ComputeNode {
 resource cpu1_0:CPU { frequency = 1602.0 load = 0.0 }
 resource ram1:RAM { total = 10079.0 free = 10079.0 }
 resource disk1:DISK { total = 5637.0 free = 5637.0 }
 resource network1:NETWORK { latency = 298.0 throughput = 45704.0
 }
}
resource resource2:ComputeNode {
 resource cpu2_0:CPU { frequency = 976.0 load = 0.0 }
 resource ram2:RAM { total = 3461.0 free = 3461.0 }
 resource disk2:DISK { total = 7222.0 free = 7222.0 }
 resource network2:NETWORK { latency = 22.0 throughput = 94130.0 }
}
property total [MB]
property free [MB]
meta size
property energy [J]
property quality [%]
component component_0 {
 contract implementation_0i0 {
  requires component the_component_0i0_0 of type component_0i0_0
  requires component the_component_0i0_1 of type component_0i0_1
  requires resource compute_resource_0 of type ComputeNode with { cpu_0 of type CPU  ram_1 of type RAM  disk_1 of type DISK
    network_1 of type NETWORK }
  requiring the_component_0i0_0.quality ≥ 100.0
  requiring the_component_0i0_1.quality ≥ 71.0
  requiring cpu_0.frequency ≥ 1858.0
  requiring ram_1.total ≥ 2825.0
  requiring disk_1.total ≥ 2707.0
  requiring network_1.throughput ≥ 5414.0
  providing quality = 6.0
  providing energy = ((0.87*(size^2.0))+(0.98*cpu_0.frequency))
 }
 contract implementation_0i1 {
  requires component the_component_0i1_0 of type component_0i1_0
  requires component the_component_0i1_1 of type component_0i1_1
  // other values
 }
 using property quality
 using property energy
}

component component_0i0_0 {
 contract implementation_0i0_0i0 { /* requirements */ }
 contract implementation_0i0_0i1 { /* requirements */ }
 using property quality
 using property energy
}

component component_0i0_1 {
 contract implementation_0i0_1i0 { /* requirements */ }
 contract implementation_0i0_1i1 { /* requirements */ }
 using property quality
 using property energy
}

component component_0i1_0 {
 contract implementation_0i1_0i0 { /* requirements */ }
 contract implementation_0i1_0i1 { /* requirements */ }
 using property quality
 using property energy
}

component component_0i1_1 {
 contract implementation_0i1_1i0 { /* requirements */ }
 contract implementation_0i1_1i1 { /* requirements */ }
 using property quality
 using property energy
}
request request0 for component_0 {
 meta size = 917.0
 requiring quality ≥ 45.0
}
minimize sum(energy)
```

Figure 4: Example input model

```
Minimize
 energy: + 714771.27 request0#implementation_0i1#resource0 + 261748.93 request0#implementation_0i1_0i1#resource1 + 564069.07 request0#
       implementation_0i1_1i0#resource2

Subject To
 request0_opc_component_0: + request0#implementation_0i1 + request0#implementation_0i0 ≤ 1
 request0_opc_component_0i0_0: + request0#implementation_0i0_0i1 + request0#implementation_0i0_0i0 ≤ 1
 request0_opc_component_0i0_1: + request0#implementation_0i0_1i1 + request0#implementation_0i0_1i0 ≤ 1
 request0_opc_component_0i1_0: + request0#implementation_0i1_0i1 + request0#implementation_0i1_0i0 ≤ 1
 request0_opc_component_0i1_1: + request0#implementation_0i1_1i1 + request0#implementation_0i1_1i0 ≤ 1
 request0_single_implementation_0i1: - request0#implementation_0i1 + request0#implementation_0i1#resource0 = 0
 request0_single_implementation_0i1_0i1: - request0#implementation_0i1_0i1 + request0#implementation_0i1_0i1#resource1 = 0
 request0_single_implementation_0i1_1i0: - request0#implementation_0i1_1i0 + request0#implementation_0i1_1i0#resource2 = 0
 request0_implementation_0i0_req_component_0i0_0: - request0#implementation_0i0 + request0#implementation_0i0_0i1 + request0#
       implementation_0i0_0i0 ≥ 0
 request0_implementation_0i0_req_component_0i0_1: - request0#implementation_0i0 + request0#implementation_0i0_1i1 + request0#
       implementation_0i0_1i0 ≥ 0
 request0_implementation_0i1_req_component_0i1_0: - request0#implementation_0i1 + request0#implementation_0i1_0i1 + request0#
       implementation_0i1_0i0 ≥ 0
 request0_implementation_0i1_req_component_0i1_1: - request0#implementation_0i1 + request0#implementation_0i1_1i1 + request0#
       implementation_0i1_1i0 ≥ 0
 request0_implementation_0i1_reqs_quality_from_component_0i1_0: - 18 request0#implementation_0i1#resource0 + 30 request0#
       implementation_0i1_0i1#resource1 ≥ 0
 request0_implementation_0i1_reqs_quality_from_component_0i1_1: - 75 request0#implementation_0i1#resource0 + 75 request0#
       implementation_0i1_1i0#resource2 ≥ 0
 request0_target: + request0#implementation_0i1 + request0#implementation_0i0 = 1
 request0_req_quality: + 45 request0#implementation_0i1#resource0 ≥ 45
 one_on_resource2: + request0#implementation_0i1_1i0#resource2 ≤ 1
 one_on_resource0: + request0#implementation_0i1#resource0 ≤ 1
 one_on_resource1: + request0#implementation_0i1_0i1#resource1 ≤ 1
```

Figure 5: ILP for the example model with Bounds and Generals sections left out

```
solution {
 request0 -> implementation_0i1 {
  compute_resource_0 -> resource0 {
   cpu_0 -> cpu0_0
   ram_1 -> ram0
   disk_1 -> disk0
   network_1 -> network0
  }
  the_component_0i1_0 -> implementation_0i1_0i1 {
   compute_resource_0 -> resource1 {
    cpu_0 -> cpu1_0
    ram_1 -> ram1
    disk_1 -> disk1
    network_1 -> network1
   }
  }
  the_component_0i1_1 -> implementation_0i1_1i0 {
   compute_resource_0 -> resource2 {
    cpu_0 -> cpu2_0
    ram_1 -> ram2
    disk_1 -> disk2
    network_1 -> network2
}}}}
```

Figure 6: Solution of the example input model

Table 1: Solving time and solution quality

Scenario	Solver	Found solution	Optimal?	Generation time (ms)	Solving time (ms)
0 (trivial)	direct	yes	yes	0	0
	external	yes	yes	0	
1 (small)	direct	yes	yes	4	0
	external	yes	yes	1	9
2 (small,	direct	yes	yes	7	0
much hardware)	external	yes	yes	3	15
3 (small,	direct	yes	yes	98	5
complex software)	external	yes	yes	64	164
4 (medium)	direct	yes	no	1400	timeout
	external	yes	no	668	timeout
5 (medium,	direct	yes	no	5389	timeout
much hardware)	external	no	–	timeout	–
6 (medium,	direct	no	–	35810	timeout
complex software)	external	no	–	timeout	–
7 (large)	direct	no	–	8518	timeout
	external	no	–	timeout	–

4 Conclusion

In this paper, we detailed the reference implementation of the TTC 2018 case. We used JastAdd to transform the input model into an intermediate ILP model, which is used in two slightly different ways by GLPK, an off-the-shelf ILP solver. Using this approach, we could get valid solutions for the first six scenarios, out of which four are optimal. However, for a real world application, our achieved solving times are too high. If an optimal solution is not required, heuristic approaches may be more adequate for this case.

Acknowledgements

This work has been funded by the German Research Foundation within the Collaborative Research Center 912 Highly Adaptive Energy-Efficient Computing, within the Research Training Group Role-based Software Infrastructures for continuous-context-sensitive Systems (GRK 1907) and the research project "Rule-Based Invasive Software Composition with Strategic Port-Graph Rewriting" (RISCOS) and by the German Federal Ministry of Education and Research within the project "OpenLicht".

References

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd system—modular extensible compiler construction. *Science of Computer Programming*, 69(1):14–26, 2007.

[GMSA18] Sebastian Götz, Johannes Mey, Rene Schöne, and Uwe Aßmann. Quality-based software-selection and hardware-mapping as model transformation problem. *11th Transformation Tool Contest (TTC 2018)*, 2018.

[Hed00] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.