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

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

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 an extended resource allocation problem [GMSA18]. A system is described by its software components with variants and component dependencies, typed hardware resources, and requests for certain software components. 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 tree for each request, with pairs of component implementation variants and assigned resources as nodes and component dependencies as edges. Each mapping has to obey the constraints defined for each software variant, i.e., it has to fulfil its *contract*. Further, a mapping is optimal, if the value of the specified objective function is optimal. The value of the objective function depends on the selection of software variants, as well as on the resources they are 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].

2 Transformation into a Linear Program

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 reference attribute grammars (RAGs) [Hed00] adding computations to nodes of the model. To solve the task, the given problem formulation is transformed into an integer linear program utilizing an intermediate representation. The remainder of this section describes the transformation problem in detail, first describing ILPs and the target model in Section 2.1, then the transformation in Section 2.2, and finally giving a detailed example in Section 2.3.

[1] `https://git-st.inf.tu-dresden.de/stgroup/ttc18` within the module `jastadd-mquat-solver-ilp`

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 11th Transformation Tool Contest, Toulouse, France, 29-06-2018, published at http://ceur-ws.org

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

IlpObjective ::= <Kind:IlpObjectiveKind> IlpLeftHandSide ; // IlpObjectiveKind is either MINIMIZE or MAXIMIZE
IlpConstraint ::= <Name:String> IlpLeftHandSide <ClauseComparator:ClauseComparator> <RightHandSide:double> ;
IlpLeftHandSide ::= IlpTerm* ;
IlpTerm ::= <Value:double> <Ref:IlpVariable> ;

abstract IlpVariable ::= <Name:String> <Request:Request> <Impl:Implementation> <Illegal:boolean> ;
IlpAllResourcesVariable:IlpVariable ;
IlpMappingVariable:IlpVariable ::= <Resource:Resource> ;
```

Listing 1: The target model describing an ILP.

```
1  for (Request request : this.getRequestList()) {
2    for (Implementation impl : comp.getImplementationList()) {
3      for (Clause rqClause : impl.requirementClauses()) {
4        if (rqClause.getDesignator().isSoftwareDesignator()) {
5          IlpLeftHandSide reqLhs = new IlpLeftHandSide();
6          for (Tuple<Implementation, Clause> tuple : rqClause.providingClauses()) {
7            Implementation prImpl = tuple.getKey(); Clause prClause = tuple.getValue();
8            for (Resource res : this.getHardwareModel().getResourceList())
9              reqLhs.addIlpTerm(new IlpTerm(prClause.evalUsing(request, res), getIlpVariable(request, prImpl, resource)));
10         }
11         // negate the term to move it to the right side
12         for (Resource res : this.getHardwareModel().getResourceList())
13           reqLhs.addIlpTerm(new IlpTerm(makeNegative(rqClause.evalUsing(request, res)), getIlpVariable(request, impl, res)));
14         result.addIlpConstraint(new IlpConstraint(reqLhs, rqClause.getClauseComparator(), 0));
15       }
16     }
17   }
18 }
```

Listing 2: Negotiation constraint computation.

2.1 Structure of the Linear Program

An ILP is described by a set of constraints over variables with integer values. Its canonical form is

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

where $\mathbf{b}$ and $\mathbf{c}$ are vectors, A is a coefficient matrix, and $\mathbf{x}$ is the solution vector of integer variables. The intermediate model used in this solution is described by the grammar shown in Listing 1. It describes a restricted set of linear programs required in this case: variables are assumed to be binary, so no bounds have to be specified.

The basic idea is to use binary variables for 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. The following constraints are created.

1. **Structural Constraints** ensure the correct structure and the functional properties of the solution.
   (a) For every request, there is exactly one variant of the target component chosen.
   (b) For every component and request pair, there is at most one variant chosen.
   (c) If a component is selected, its required components must also be selected.
   (d) On every resource, there can be at most one deployed component.
2. **Contract Negotiation Constraints** ensure non-functional requirements of requests and components.
   (a) The non-functional properties of a request must be met.
   (b) The non-functional requirements of required components must be met.

Listing 2 shows the computation of contract negotiation constraint 2b and Listing A shows an example ILP.

2.2 ILP Transformation using Higher Order Attributes

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. JastAdd attributes are exposed and invoked like ordinary Java methods, e.g., in line 6 of Listing 2. We defined 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

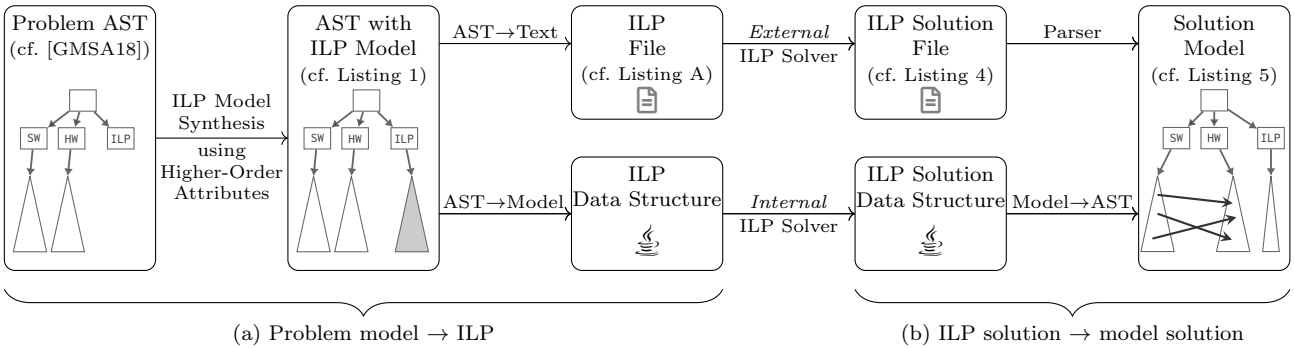

Figure 1: Models and transformations in the solving process.

transformation of the source model to the ILP model. The main motivation for using attributes is the ability to cache their computed values. With that, clauses need only be evaluated once for a certain configuration. The given problem model is transformed into the intermediate model shown in Listing 1 using higher-order attributes [VSK89]: the ILP model is defined as a subtree of the problem model and is computed by an attribute.

Once the intermediate ILP model is generated, there are two possible ways to get a solution for the ILP. The first, *external* way is to transform the model to a textual representation and invoke an external solver which in turn outputs a solution file. Here, the overhead of reading and writing files arises, which can be significant for larger models. We use GLPK[2] as external solver, but the employment of a standardized ILP notation allows the use of other solvers as well. The second, *internal* 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, which 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 creating an assignment for each of them. An overview over the entire transformation process from the problem to the solution model is given in Figure 1.

2.3 Example Model and Solution

To illustrate the process, consider the model defined in Listing 3. There are three resources, one main component targeted by the request and requiring two other components. All components have two implementations each. The *external* ILP solver transforms this model to the ILP shown in Listing A. One might expect to see more variables in the objective function (line 2). However, during the transformation, some illegal assignments with respect to resource and quality requirements are already discarded, and, thus not show up in the ILP. Further, the constraints can be seen in lines 4-36. Finally, the variables have to be declared as binary variables in the `Binaries` section in lines 38-54. In this case, the solver finishes successfully, finding an optimal solution with an objective value of 23284.15. Listing 4 shows an extract of the solver output. All variables with a value of 1 are used to construct the solution as shown in Listing 5.

3 Evaluation

To investigate the feasibility of the approach, the implementation was tested with the scenarios provided in the case description [GMSA18]. The measurements were performed on an Intel Xeon E5-2643 machine with 32G of memory using Ubuntu 16.04 with GLPK 4.65 and Oracle Java 1.8.0_181. A maximum solving time of 15 minutes per run was allowed; each scenario was executed five times with both solver variants. Table 1 shows the median results of the runs. The test results are only shown for the small, medium and large scenario sizes, since on the given hardware, it was not possible to find valid solutions for the huge scenarios within the given time. The table also shows whether a valid solution has been found and whether the ILP solver can guarantee its optimality. 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 *total solving time* specifies the total time it took to generate

[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 *internal* solver needs about 120 lines of code (excluding logging).

```
container resource type ComputeNode { /* type definitions */ }

resource res0:ComputeNode {
  resource cpu0_0:CPU { frequency = 1034.0 /* ... */ }
  resource ram0:RAM { total = 13409.0 /* ... */ }
  resource disk0:DISK { total = 12256.0 /* ... */ }
  resource network0:NETWORK { throughput = 54883.0 /* ... */ }
}
resource res1:ComputeNode { /* ... */ }
resource res2:ComputeNode { /* ... */ }

property total [MB]
property free [MB]
meta size
property energy [J]
property quality [%]
component component_0 {
  contract impl_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 ≥ 1.0
    requiring the_component_0i0_1.quality ≥ 8.0
    requiring cpu_0.frequency ≥ 2245.0
    requiring ram_1.total ≥ 14608.0
    requiring disk_1.total ≥ 7308.0
    requiring network_1.throughput ≥ 23804.0
    providing quality = 16.0
    providing energy = ((0.1*(size^2.0))+(0.82*cpu_0.frequency))

  }
  contract impl_0i1 { /* ... */ }
  using property quality
  using property energy
}
component component_0i0_0 { /* ... */ }
component component_0i0_1 { /* ... */ }
component component_0i1_0 { /* ... */ }
component component_0i1_1 { /* ... */ }

request rq0 for component_0 {
  meta size = 147.0
  requiring quality ≥ 18.0
}
minimize sum(energy)
```

Listing 3: Example input model (shortened).

```
Objective:  obj = 23284.15 (MINimum)

Column name                 Activity
--------------------------- --------
rq0#impl_0i0_1i0#resource1  0
rq0#impl_0i1_0i0#resource1  1
rq0#impl_0i1_0i1#resource1  0
rq0#impl_0i1#resource0      1
rq0#impl_0i1_1i0#resource2  1
rq0#impl_0i1_1i0#resource0  0
rq0#impl_0i0_1i0            0
rq0#impl_0i0_1i1            0
rq0#impl_0i1_0i0            1
rq0#impl_0i1_0i1            0
rq0#impl_0i0                0
rq0#impl_0i0_0i0            0
rq0#impl_0i0_0i1            0
rq0#impl_0i1                1
rq0#impl_0i1_1i0            1
rq0#impl_0i1_1i1            0
```

Listing 4: ILP solution (extract).

```
solution {
  rq0 -> impl_0i1 {
    compute_resource_0 -> res0 {
      cpu_0 -> cpu0_0
      ram_1 -> ram0
      disk_1 -> disk0
      network_1 -> network0
    }
    the_component_0i1_0 -> impl_0i1_0i0 {
      compute_resource_0 -> res1 {
        cpu_0 -> cpu1_0
        ram_1 -> ram1
        disk_1 -> disk1
        network_1 -> network1
      }
    }
    the_component_0i1_1 -> impl_0i1_1i0 {
      compute_resource_0 -> res2 {
        cpu_0 -> cpu2_0
        ram_1 -> ram2
        disk_1 -> disk2
        network_1 -> network2
      }
    }
  }
}
```

Listing 5: Reconstructed solution.

and solve the ILP. If a step took 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 quickly. One the other hand, if the problem size and complexity increases, finding valid and optimal solutions is very hard. However, in many cases the ILP solver is able to provide valid, but not (guaranteed) optimal solutions if it is stopped prematurely by the time limit. The acquired measurements allow some further observations with regard to the two presented variants:

- For larger problems, the generation time is significantly lower than the solving time. Thus, a better formulation of the ILP or the selection of a better performing solver may lead to improved results.
- Comparing the *internal* solver which creates the ILP programmatically to the *external* one which writes the ILP into a file and then calls the external solver, both generation variants perform similarly. However, the latter variant requires additional time for reading the ILP from file.

4 Conclusion and Future Work

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 got valid solutions for eight of the 13 provided scenarios, six

Table 1: Solving time and solution quality (internal/external solver).

	Scenario	Valid?	Optimal?	Generation time (ms)	Total solving time (ms)
0	trivial	✓	✓	16 / 8	19 / 20
1	small	✓	✓	30 / 24	33 / 40
2	small, much hardware	✓	✓	40 / 33	43 / 51
3	small, complex software	✓	✓	306 / 326	333 / 491
4	medium	✓	✓/✗	2,033 / 2,168	83,742 / timeout
5	medium, much hardware	✓	✓/✗	7,134 / 6,838	84,587 / timeout
6	medium, complex software	✗	✗	42,434 / 42,686	timeout / timeout
7	large	✗/✓	✗	10,796 / 11,045	timeout / timeout
8	large, much hardware	✓	✗	37,242 / 36,661	timeout / timeout
9	large, complex software	✗	✗	538,530 / timeout	timeout / timeout

of which are optimal. However, for many use cases, our achieved solving times may be too long. If an optimal solution is not required, heuristic approaches may be more adequate for this case.

Finally, the presented solution offers some opportunities for improvement. A more advanced RAG-based analysis using partial contract evaluation or abstract contract interpretation could decrease both size and complexity of the ILP. Additionally, using a better solver and a meta-optimization of its parameters could be beneficial.

Acknowledgements

This work has been funded by the German Research Foundation within the Collaborative Research Center 912 Highly Adaptive Energy-Efficient Computing, 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. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, June 2018.

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

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 131–145, New York, NY, USA, 1989. ACM.

Appendix

```
\ Integer Linear Program in the CPLEX LP Format
\ Format Description: https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/FileFormats/topics/LP.html


\ Specification of the Objective
Minimize
energy:  +10961.72 rq0#impl_0i0_1i0#res1 +4138.75 rq0#impl_0i1_0i0#res1 +7011.62 rq0#impl_0i1_0i1#res1 +1296.54 rq0#impl_0i1#res0 +17848.86
      rq0#impl_0i1_1i0#res2 +17969.64 rq0#impl_0i1_1i0#res0


\ Constraints
Subject To

\ Define implementation variables, specifying if an implementation is selected, i.e, if a resource is mapped to the implementation.
\ Resources violating a contract are omitted.
rq0_single_impl_0i0:      - rq0#impl_0i0 = 0.0                     \  if no resources are valid, the implementation variable is 0 (false)
rq0_single_impl_0i1:      + rq0#impl_0i1#res0 - rq0#impl_0i1 = 0.0
rq0_single_impl_0i0_0i0:  - rq0#impl_0i0_0i0 = 0.0
rq0_single_impl_0i0_0i1:  - rq0#impl_0i0_0i1 = 0.0
rq0_single_impl_0i0_1i0:  + rq0#impl_0i0_1i0#res1 - rq0#impl_0i0_1i0 = 0.0
rq0_single_impl_0i0_1i1:  - rq0#impl_0i0_1i1 = 0.0
rq0_single_impl_0i1_0i0:  + rq0#impl_0i1_0i0#res1 - rq0#impl_0i1_0i0 = 0.0
rq0_single_impl_0i1_0i1:  + rq0#impl_0i1_0i1#res1 - rq0#impl_0i1_0i1 = 0.0
rq0_single_impl_0i1_1i0:  + rq0#impl_0i1_1i0#res2 + rq0#impl_0i1_1i0#res0 - rq0#impl_0i1_1i0 = 0.0
rq0_single_impl_0i1_1i1:  - rq0#impl_0i1_1i1 = 0.0

\ Structural Constraint 1a: ensure the request is fulfilled
rq0_target:  + rq0#impl_0i0 + rq0#impl_0i1 = 1.0

\ Structural Constraint 1b: Choose at most one variant per component and request.
rq0_opc_component_0:      + rq0#impl_0i0 + rq0#impl_0i1 <= 1.0
rq0_opc_component_0i0_0:  + rq0#impl_0i0_0i0 + rq0#impl_0i0_0i1 <= 1.0
rq0_opc_component_0i0_1:  + rq0#impl_0i0_1i0 + rq0#impl_0i0_1i1 <= 1.0
rq0_opc_component_0i1_0:  + rq0#impl_0i1_0i0 + rq0#impl_0i1_0i1 <= 1.0
rq0_opc_component_0i1_1:  + rq0#impl_0i1_1i0 + rq0#impl_0i1_1i1 <= 1.0

\ Structural Constraint 1c: ensure the required components of a selected component are also selected
rq0_impl_0i0_req_component_0i0_0:  + rq0#impl_0i0_0i0 + rq0#impl_0i0_0i1 - rq0#impl_0i0 >= 0.0
rq0_impl_0i0_req_component_0i0_1:  + rq0#impl_0i0_1i0 + rq0#impl_0i0_1i1 - rq0#impl_0i0 >= 0.0
rq0_impl_0i1_req_component_0i1_0:  + rq0#impl_0i1_0i0 + rq0#impl_0i1_0i1 - rq0#impl_0i1 >= 0.0
rq0_impl_0i1_req_component_0i1_1:  + rq0#impl_0i1_1i0 + rq0#impl_0i1_1i1 - rq0#impl_0i1 >= 0.0

\ Structural Constraint 1d: At most one component per resource.
one_on_res0:  + rq0#impl_0i1#res0 + rq0#impl_0i1_1i0#res0 <= 1.0
one_on_res1:  + rq0#impl_0i0_1i0#res1 + rq0#impl_0i1_0i0#res1 + rq0#impl_0i1_0i1#res1 <= 1.0
one_on_res2:  + rq0#impl_0i1_1i0#res2 <= 1.0

\ Contract Negotiation Constraint 2a: Non-functional property of the request.
rq0_req_quality:  +37.0 rq0#impl_0i1#res0 >= 18.0

\ Contract Negotiation Constraint 2b: Non-functional properties of required components
rq0_impl_0i0_reqs_quality_from_component_0i0_1:  +14.0 rq0#impl_0i0_1i0#res1 >= 0.0
rq0_impl_0i1_reqs_quality_from_component_0i1_0:  +99.0 rq0#impl_0i1_0i0#res1 +65.0 rq0#impl_0i1_0i1#res1 -65.0 rq0#impl_0i1#res0 >= 0.0
rq0_impl_0i1_reqs_quality_from_component_0i1_1:  +94.0 rq0#impl_0i1_1i0#res2 +94.0 rq0#impl_0i1_1i0#res0 -94.0 rq0#impl_0i1#res0 >= 0.0


\ Variable Definitions
Binaries \ here, all variables are binary
rq0#impl_0i0 rq0#impl_0i0_0i0 rq0#impl_0i0_0i1 rq0#impl_0i0_1i0 rq0#impl_0i0_1i0#res1 rq0#impl_0i0_1i1 rq0#impl_0i1 rq0#impl_0i1#res0
      rq0#impl_0i1_0i0 rq0#impl_0i1_0i0#res1 rq0#impl_0i1_0i1 rq0#impl_0i1_0i1#res1 rq0#impl_0i1_1i0 rq0#impl_0i1_1i0#res0
      rq0#impl_0i1_1i0#res2 rq0#impl_0i1_1i1

End
```

Listing A: ILP for the example model (sorted and commented).