SINTEF

SINTEF ICT

Address: P.O.Box 124, Blindern
0314 Oslo NORWAY
Location: Forskningsveien 1
0373 Oslo
Telephone: +47 22 06 73 00
Fax: +47 22 06 73 50

Enterprise No.: NO 948 007 029 MVA

SINTEF REPORT

TITLE

Optimizing Alloy Models

AUTHOR(S)

Andreas Svendsen, Øystein Haugen, Birger Møller-Pedersen

CLIENT(S)

Research Project MoSiS and Research Project VERDE

REPORT NO.	CLASSIFICATION	CLIENTS REF.	
SINTEF A21094	OPEN	Research Council of Norway project nr. 180110/I40 and 193264/I40	
CLASS. THIS PAGE	ISBN	PROJECT NO.	NO. OF PAGES/APPENDICES
OPEN	978-82-14-04996-1	90B246 and 90B274	15
ELECTRONIC FILE CODE		PROJECT MANAGER (NAME, SIGN.)	CHECKED BY (NAME, SIGN.)
N/A		Øystein Haugen	Arnor Solberg
FILE CODE	DATE	APPROVED BY (NAME, POSITION, SIGN.)	
N/A	2011-11-10	Bjørn Skjellaug, Research Director	

ABSTRACT

This paper presents three possible optimizations of Alloy models, including how and when to implement these optimizations. Alloy is a formal light-weight language for performing incremental and automatic analysis. Analysis is performed within a user-defined scope, which limits the number of model elements that are considered. When this scope increases, the number of possible combinations of model elements increases exponentially. Thus the analysis time escalates rapidly caused by this state-space explosion. Implementing the optimizations presented in this paper will decrease the analysis time, and thus make analysis suitable for larger models. We give concrete examples showing the decrease in analyzation effort and time given these optimizations.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	Modeling	Modellering
GROUP 2	Software analysis	Programvareanalyse
SELECTED BY AUTHOR	Optimization	Optimalisering
	Alloy	Alloy
	Train Control Language	Train Control Language

Optimizing Alloy Models

Andreas Svendsen[1,2], Øystein Haugen[1], Birger Møller-Pedersen[2]

[1]SINTEF, Pb. 124 Blindern, 0314 Oslo, Norway
[2]Department of Informatics, University of Oslo, Pb. 1080 Blindern, 0316 Oslo, Norway
andreas.svendsen@sintef.no, oystein.haugen@sintef.no, birger@ifi.uio.no

Abstract. This paper presents three possible optimizations of Alloy models, including how and when to implement these optimizations. Alloy is a formal light-weight language for performing incremental and automatic analysis. Analysis is performed within a user-defined scope, which limits the number of model elements that are considered. When this scope increases, the number of possible combinations of model elements increases exponentially. Thus the analysis time escalates rapidly caused by this state-space explosion. Implementing the optimizations presented in this paper will decrease the analysis time, and thus make analysis suitable for larger models. We give concrete examples showing the decrease in analyzation effort and time given these optimizations.

Keywords. Alloy, analysis, optimization, modeling, Train Control Language.

1 Introduction

Formal analysis of software systems is mainly applied to prove the correctness of certain properties of systems. Traditionally, this has required extended knowledge of mathematical techniques, including mathematical notation and theorem proving. Thus, this has been a time-consuming and tedious process with a high cost. Therefore, even though such analysis can give valuable findings, especially when applied on model level in the initial phase of developing software, it is not as widely used, as it ought to be.

There is tool support for automating the analysis of software systems. Alloy is a structural modeling language to define software systems precisely through a set of structural constraints. Such software systems are often defined by domain-specific languages (DSL). The Alloy Analyzer is a tool that supports incremental and fully automatic analysis of Alloy models, giving immediate feedback in form of a solution model that satisfies the constraints specified in the Alloy model. This is performed by transforming the Alloy model to first-order logic constraints and using a SAT-solver [6] to perform the analysis. Alloy grows in popularity due to its automatic analyzer and its uncomplicated notation. Kelsen and Ma [8] have shown that Alloy offers a more uniform notation for performing analysis than traditional formal methods.

Alloy is a light-weight declarative language and the Alloy Analyzer only guarantees for the result up to a user-given scope, which specifies the number of possible model elements in the solution model. Analysis is performed immediately when the analyzed model is small. However, when the model grows, the scope must be increased in order to be able to perform analysis on the model. An increased scope implies that the number of possible combinations of model elements increases exponentially. Thus, the analysis time escalates rapidly and the analysis is, in the best case, not performed immediately. In the worst case, the analyzer can run out of resources (i.e. computer memory) in the search for a solution.

We have used Alloy to formalize the Train Control Language (TCL) which is a DSL for modeling train stations. Analysis of the station models, such as simulations with a given number of trains, is important since TCL represents a safety-domain. Performing such analysis on relative small station models turned out to be time-consuming and unfeasible when the number of trains increased in the analysis. An investigation of the Alloy models implied that Alloy allows certain forms for optimizations that can be very profitable. These optimizations are not specific for TCL, but allow more efficient analysis of any kinds of systems.

One obvious optimization of Alloy models is to lower the scope, and thus decreasing the possible number of objects. Furthermore, the scope can be specified individually for each kind of object in the model, such that only the required number of each kind of object is considered. However, this is not always feasible. There are models that require larger scopes (e.g. TCL models), and thus become time-consuming or unfeasible to analyze. Another optimization is to use a platform-specific SAT-solver (e.g. minisat [2, 10]). However, even though this can improve the analysis time, large Alloy models still require other optimizations to reduce the analysis time sufficiently.

In this paper we present three optimizations of Alloy models that can reduce the analysis time substantially. The contribution of the paper is therefore the description of these three optimizations, when and how to implement them, and their application on concrete TCL examples. We show the decrease in analysis effort and time when applying the optimizations on these models. As the example illustrates, the presented optimizations are useful when developing and analyzing models in DSLs using Alloy.

The outline of the paper is as follows: Section 2 gives further background information about Alloy and the train control language TCL. Section 3 elaborates further on the challenges with analyzing TCL models and the analysis effort required. Section 4 describes the three optimizations, and Section 5 gives a discussion of the application of such optimizations. Section 6 presents related work, while Section 7 finally gives some concluding remarks and future work.

2 Background

Before discussing the optimization of Alloy models, we give some background information. First, we further explain the Alloy language and what kinds of analysis that can be performed automatically by the Alloy Analyzer. Then we introduce the

example from the train domain and explain how this example benefits from automatic analysis using Alloy.

2.1 Alloy

Alloy is a declarative light-weight language for formally modeling a software system, such that analysis can be automatically conducted by its analyzer tool. The Alloy Analyzer takes an Alloy model as input, searches for a solution satisfying the constraints in the Alloy model, and gives a solution model as output. Unlike traditional theorem proving and model-checking, the Alloy Analyzer only performs analysis up to a user-specified scope. This scope limits the number of instantiated model elements such that analysis can be performed efficiently. The scope can even be set to restrain the number of each kind of model element independently.

An Alloy model usually consists of *signatures*, *fields*, *facts*, *predicates* and *assertions*. A signature defines a type in the model, much like a *class* in object-oriented languages, which can be instantiated into model elements (objects). A signature can extend another signature to form a hierarchy of signatures. Furthermore, a signature can contain fields, which refer other signatures, much like *associations* or *references* in object-oriented languages. A fact consists of a set of global constraints that must be satisfied for all kinds of analysis performed on the model. A predicate consists of a set of constraints that must be satisfied if the predicate is included in the analysis, much like an operation in object-oriented languages. An assertion is a set of constraints that are claimed to be satisfied for certain kinds of analysis.

The Alloy Analyzer allows two kinds of analysis to be performed on Alloy models: Search for a solution model that satisfies a predicate or search for a counter-example that falsifies an assertion. In both cases, the analyzer will populate each signature, with corresponding fields, with model elements up to the user-specified scope, and try all possible combination until all constraints are satisfied. If a solution is found, the analyzer returns a *solution model* providing the solution. This solution model is an arbitrary solution (or counter-example) and is not guaranteed to be optimal or to have any other properties.

The Alloy Analyzer is invoked by either using the provided Alloy editor or by using the provided set of APIs to integrate with Java programs. We will use the Alloy editor to illustrate the optimizations presented in Section 4. The Alloy model is transformed into a Kodkod model, which is a SAT-based model finder [9, 15], and a SAT-solver [6] is used for calculating the solution. The solution is then transformed back into a solution model in Alloy. We will use the SAT-solver MiniSat [10], which is platform-specific, to perform the analysis in this paper, since it is supplied with the Alloy tool and is more efficient than SAT4J [11], which is platform-independent. However, the optimizations discussed in this paper are Alloy-specific, and do not depend on a particular SAT-solver.

An example of concrete Alloy syntax is shown in Fig. 1. This example is taken from the train domain introduced in the next section. A *Track* represents a physical part of a train station with a *start* and *end*. This is modeled by having a signature Track with two fields, start and end, where each of the fields refers another signature, *Endpoint*. Note that each field can only refer *one* Endpoint object. Furthermore, a fact

is specifying that *no* tracks exist such that the two fields, start and end, refer the same Endpoint object.

```
abstract sig Track {
    start: one Endpoint,
    end: one Endpoint
}

fact {
    no t:Track, e:t.start, e2:t.end | e in e2
}
```

Fig. 1. Definition of a Track in Alloy

2.2 Train Control Language

The Train Control Language (TCL) is a DSL for modeling train signaling systems [3, 13]. A DSL captures the concepts of a particular domain and how they are related by using a metamodel. A metamodel is a model that can be instantiated into models, describing applications within the domain. Accordingly, the TCL metamodel defines the concepts of a train station and how these concepts are related. This metamodel can be instantiated into particular train station models that represent the physical train stations. The models can be used to generate code using a model transformation. TCL models are e.g. used as the basis for generating tables, verification documents and source code for controlling the signaling system on the station. For further information about DSL and model transformation technology, we refer to [7].

An excerpt of the TCL metamodel, illustrating the most important concepts and their relations, is illustrated in Fig. 2. A *Station* is the top concept containing the other concepts. A *TrainRoute* is a route into or out of a Station, which must be reserved by a train before the train can move. Each TrainRoute refers a set of *TrackCircuits*, which model a part of a station where the train can be located. A TrackCircuit refers a set of *Tracks*, which can either be a *LineSegment* or a *Switch*. These Tracks are connected through *Endpoints*, which are again connected to a *Signal*. The concrete syntax of TCL is illustrated in Fig. 3. Note that TCL also consists of other elements than the ones illustrated, such as *Stillers*, *Buildings*, *Derailers*, etc., but for simplicity we have omitted them in this paper. For more information about TCL, we refer to [3, 13].

To perform analysis on TCL models, we have formalized TCL using Alloy. This yields four Alloy models: *Static semantics*, *dynamic semantics*, *instance specification* and *analysis model*. The static semantics model defines the structure of the language and is basically transformed from the metamodel. The dynamic semantics model defines the behavior of TCL, meaning how trains can move into and out of a station. The instance specification model extends the static semantics model to define a particular station model. The analysis model defines a set of predicates and assertions to simulate and check certain properties of station models.

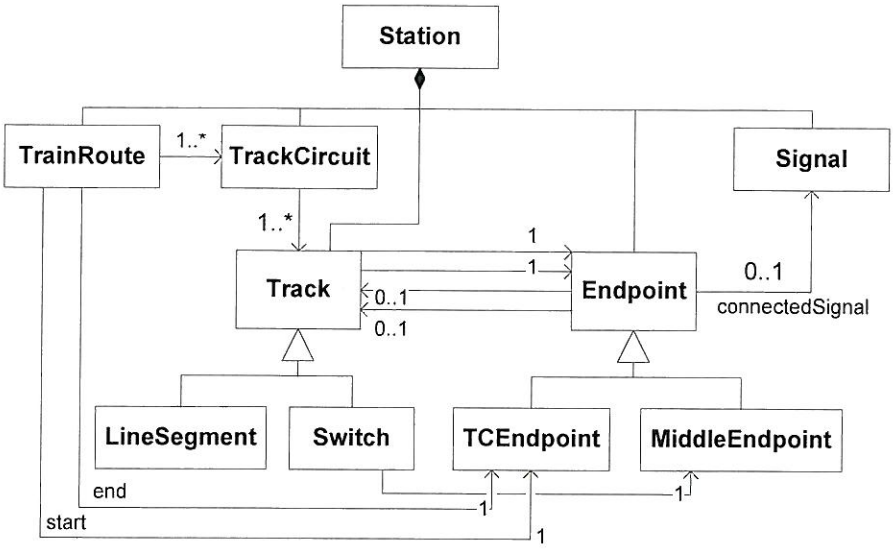

Fig. 2. Excerpt of the TCL metamodel

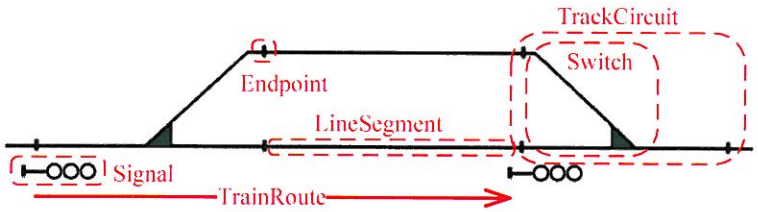

Fig. 3. TCL concrete syntax annotated with TCL concepts (in red)

More specifically, the static semantics model contains a set of signatures, fields and global constraints for the TCL language. The instance specification extends these signatures to represent each object in the particular instance model. Constraints are used to specify the links between the extended signatures. Thus, the instance specification is restricting the TCL static semantics to only one instance model. The dynamic semantics model contains predicates and facts to define the dynamic behavior of TCL. Intuitively, the behavior consists of three operations: Introduce a new train on either side of the station, reserve a TrainRoute for a train, and move a train one TrackCircuit ahead. The kinds of analysis performed on TCL models include simulation of the station with a particular number of trains, checking for the maximum number of trains allowed simultaneously, checking whether TrainRoutes can be allocated when certain TrackCircuits are occupied, etc. For more information about the formal representation of TCL, we refer to [12].

3 Problem Description

Before describing the optimizations of Alloy models, we first further discuss the challenge with state-space explosion by illustrating it using a couple of examples. We

look at two different TCL models, two-track station and three-track station, and simulation with two, three and four trains on these stations, to investigate the impact of the optimizations. This section shows the initial analysis effort and time before applying optimizations.

A typical TCL model, a two-track station, is illustrated in Fig. 4. Note that the rectangles on the top part of the figure represent TrainRoutes and TrackCircuits. In the figure there are 1 Station, 8 TrainRoutes, 6 TrackCircuits, 8 Tracks, 10 Endpoints, 6 Signals, 4 Stillers and 2 Buildings. Another TCL model is illustrated in Fig. 5. This station model has 1 Station, 12 TrainRoutes, 6 TrackCircuits, 11 Tracks, 14 Endpoints, 8 Signals, 5 Stillers and 2 Buildings.

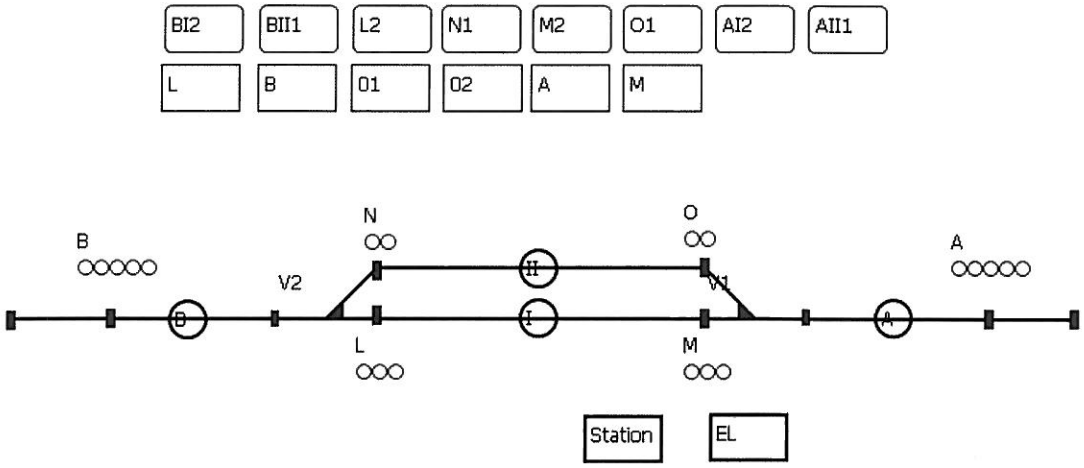

Fig. 4. TCL model – two-track station

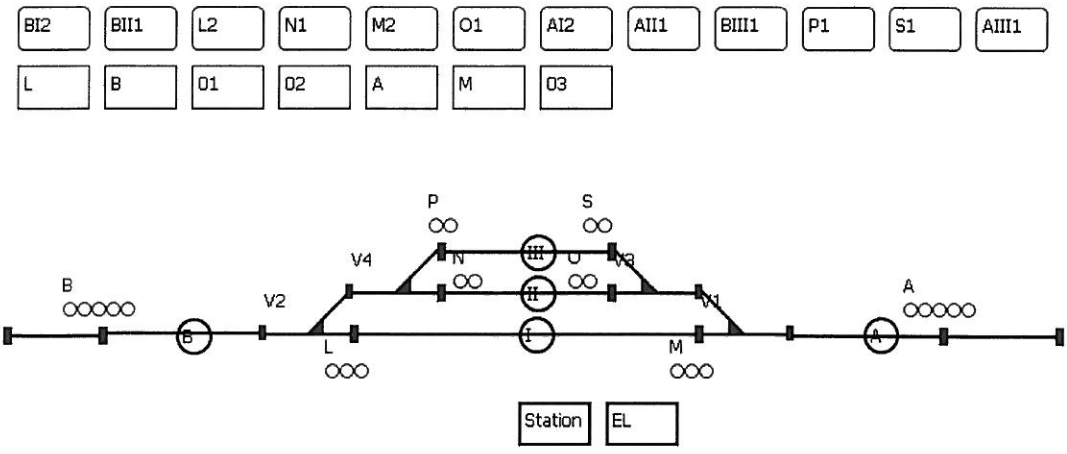

Fig. 5. TCL model – three-track station

Even though TCL includes integration to Alloy through its APIs, we have decided to use the editor provided by the Alloy tool. Then we avoid extra overhead from the TCL editor itself and the results will be more comparable with other attempts to perform analysis. The result of performing simulation with two and four trains on the two-track station (Fig. 4), using MiniSat, is illustrated in Fig. 6. Similarly, performing

the simulations on the three-track station (Fig. 5), also using MiniSat, yields the results illustrated in Fig. 7.

```
Executing "Run run$1 for exactly 19 State, 2 Train, 1 Station, 8 TrainRoute, 6 TrackCircuit, 8 Track, 10
Endpoint, 6 Signal, 4 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  140715 vars. 1870 primary vars. 424053 clauses. 25092ms.
  Instance found. Predicate is consistent. 1486ms.

Executing "Run run$1 for exactly 37 State, 4 Train, 1 Station, 8 TrainRoute, 6 TrackCircuit, 8 Track, 10
Endpoint, 6 Signal, 4 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  490335 vars. 4362 primary vars. 1718602 clauses. 100191ms.
  Instance found. Predicate is consistent. 168092ms.
```

Fig. 6. Analysis effort when simulating a two-track station

```
Executing "Run run$1 for exactly 19 State, 2 Train, 1 Station, 12 TrainRoute, 7 TrackCircuit, 11 Track, 14
Endpoint, 8 Signal, 5 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  315599 vars. 2818 primary vars. 898980 clauses. 86110ms.
  Instance found. Predicate is consistent. 7814ms.

Executing "Run run$1 for exactly 37 State, 4 Train, 1 Station, 12 TrainRoute, 7 TrackCircuit, 11 Track, 14
Endpoint, 8 Signal, 5 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  1074682 vars. 6022 primary vars. 3508706 clauses. 352068ms.
  Instance found. Predicate is consistent. 122367ms.
```

Fig. 7. Analysis effort when simulating a three-track station

The figures illustrate the number of objects of each signature in the analysis. The analysis is performed with all options set to default, except the SAT-solver, which is specified to be MiniSat. Furthermore, we see the number of variables and clauses needed to perform the analysis, the time needed to transform the Alloy model to Conjunctive Normal Form (CNF), which are Boolean formulas, and the time required to find an arbitrary solution to the formulas. Note that the time required to find a solution (solving-time) is relatively arbitrary. It can vary dependent on the kind of SAT-solver used, what kinds of constraints that restrict the model and what kind of analysis that are performed. However, the number of variables and clauses define the complexity of the Boolean formulas, and can therefore be a good measurement for improvement. We will therefore use these to measure the optimizations in Section 4, but we also advice noticing the analysis time, since it gives an indication. Note that the simulation of the three-track station model with four trains requires more than 1M variables and 3.5M clauses (and about 8 minutes).

4 Optimizing Alloy Models

As illustrated in the last section, performing rather simple analysis requires quite a lot of analysis effort. Further increasing the size of the stations or the number of trains in the simulation will escalate the required analysis effort and time. There are huge

benefits to gain from using optimizations that are rather straightforward to implement. In this section we present three such optimizations, which consist of reducing the complexity of Alloy expressions, introducing partial instances to model fixed parts of the model being analyzed, and hinting the number of instantiated objects of each signature by using subsets.

4.1 Reducing the Complexity of Expressions

Reducing the complexity of Alloy expressions (constraints) can reduce the overall complexity of the Alloy model and thus the analysis effort required. A straightforward method for reducing the complexity is to minimize the use of unnecessary variables, meaning that variables used only in sub-expressions, should also only be defined within the sub-expressions. This will avoid populating and checking solutions where the variables have different values but are not used in other sub-expressions. An example from the TCL dynamic semantics is illustrated in Fig. 8. The constraint shown in the figure restricts all trains to be related to only one TrainRoute and TrackCircuit simultaneously. More specifically, for all states *s*, trains *t*, TrainRoutes *tr* and *tr'*, and TrackCircuits *tc* and *tc'*, if the relation between t and tr exists in s.trainOnRoute and the relation between t and tr' also exists in s.trainonRoute, then tr must be equal tr', and similar for TrackCircuits.

```
fact one_allocatedRoute_and_occupiedTC_per_train{
      all s:State, t:Train, tr, tr':TrainRoute, tc,tc':TrackCircuit {
                  t->tr in s.trainOnRoute && t->tr' in s.trainOnRoute => tr in tr'
                  t->tc in s.trainOnTrack && t->tc' in s.trainOnTrack => tc in tc'
      }
}
```

Fig. 8. Expression with six variables (s, t, tr, tr', tc and tc')

Note that there are two sub-expressions in the *all*-expression; one for TrainRoutes and one for TrackCircuits. The variables for TrackCircuit are not used by the sub-expression for TrainRoutes, and similar for the TrainRoute variables and the TrackCircuit expression. Since all variables are defined in the all-expression, all possibilities have to be populated, even though the value of the variables may not be significant (i.e. in the TrainRoute sub-expression the values for the TrackCircuit variables are not significant, but still have to be evaluated). Reducing the complexity of this fact yields the fact illustrated in Fig. 9. Note that the TrainRoute variables and TrackCircuit variables now are defined locally in the sub-expressions. Further notice that this optimization does not have any side-effects, since it simply re-factors the original expression, and should therefore be implemented in all kinds of Alloy models when possible.

The results of running the simulations presented in Section 3 with this optimization implemented are illustrated in Fig. 10 for the two-track station and in Fig. 11 for the three-track station. Note that the number of variables and the number of clauses used in the analysis have been reduced (i.e. for the three-track station simulated with 4 trains, the number of variables has reduced from 1M to 652k and the number of clauses has reduced from 3.5M to 2.6M). Note that this optimization is only

performed on one expression (fact) in the TCL dynamic semantics model. Therefore, the potential benefit from further re-factoring other such expressions can be larger than the results illustrated in this paper.

```
fact one_allocatedRoute_and_occupiedTC_per_train{
      all s:State, t:Train {
            all tr, tr':TrainRoute {
                  t->tr in s.trainOnRoute && t->tr' in s.trainOnRoute => tr in tr'
            }

            all tc,tc':TrackCircuit {
                  t->tc in s.trainOnTrack && t->tc' in s.trainOnTrack => tc in tc'
            }
      }
}
```

Fig. 9. Expression with reduced complexity

```
Executing "Run run$1 for exactly 19 State, 2 Train, 1 Station, 8 TrainRoute, 6 TrackCircuit, 8 Track, 10
Endpoint, 6 Signal, 4 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  107313 vars. 1870 primary vars. 358731 clauses. 22883ms.
  Instance found. Predicate is consistent. 3733ms.

Executing "Run run$1 for exactly 37 State, 4 Train, 1 Station, 8 TrainRoute, 6 TrackCircuit, 8 Track, 10
Endpoint, 6 Signal, 4 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  360243 vars. 4362 primary vars. 1464190 clauses. 95627ms.
  Instance found. Predicate is consistent. 109341ms.
```

Fig. 10. Analysis effort with reduced complexity for simulation of two-track station

```
Executing "Run run$1 for exactly 19 State, 2 Train, 1 Station, 12 TrainRoute, 7 TrackCircuit, 11 Track, 14
Endpoint, 8 Signal, 5 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  207109 vars. 2818 primary vars. 685154 clauses. 83110ms.
  Instance found. Predicate is consistent. 2954ms.

Executing "Run run$1 for exactly 37 State, 4 Train, 1 Station, 12 TrainRoute, 7 TrackCircuit, 11 Track, 14
Endpoint, 8 Signal, 5 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  652142 vars. 6022 primary vars. 2675910 clauses. 337256ms.
  Instance found. Predicate is consistent. 84838ms.
```

Fig. 11. Analysis effort with reduced complexity for simulation of three-track station

4.2 Partial Instances

The introduction of partial instances is one of the main benefits of the use of Kodkod as the model finder for Alloy version 4 [1]. This optimization allows the modeler to

express associations between objects using static functions. These static functions represent a particular type of a relation, from one signature to another signature, and return a union of all the relations that exist between these signatures.

The benefit of using static functions to represent associations is that the relations represented by the functions can be computed without the need to populate all possible models that do not satisfy the relations. If parts of the model being analyzed are fixed, the use of partial instances can improve the analysis effort by an order of magnitude. In general, this optimization is to be used when the structure (or parts of the structure) of the model is known. Then the Alloy Analyzer does not have to calculate all possible models and discard the invalid ones. Instead it can just select the ones that are correct. Partial instances are particularly useful when analyzing a given model instance (e.g. a two or three track station model), since the model elements and their references are fixed.

Consider the TCL example where we analyze a particular TCL model by extending the signatures of the static semantics model. Fig. 12 represents a TrainRoute with a set of fields defined by the static semantics model. This TrainRoute signature is extended to a particular TrainRoute and restricted to refer to and being referred by other objects in the model (see Fig. 13). Note that using fields to represent associations require the analysis to populate all possible solution models and discard the ones that do not satisfy the constraints representing the fields. Therefore, the analysis effort by using fields to represent associations, even though the associations are fixed, greatly escalates the analysis effort required.

```
abstract sig TrainRoute {
    trackCircuits: some TrackCircuit,
    start: one TrackCircuitEndpoint,
    end: one TrackCircuitEndpoint,
    direction: one Direction
}
```

Fig. 12. A TrainRoute defined by the static semantics (not optimized)

```
one sig tr_L2 extends TrainRoute{} {#trackCircuits = 2}
fact {all tr:tr_L2, st:Station2T | tr in st.trainRoutes}
fact {all tr:tr_L2, e:ep_TCE2 | e in tr.start}
fact {all tr:tr_L2, e:ep_StationA | e in tr.end}
fact {all tr:tr_L2, tc:tc_B | tc in tr.trackCircuits}
fact {all tr:tr_L2, tc:tc_L | tc in tr.trackCircuits}
fact {all tr:tr_L2, dir:Left | dir in tr.direction}
```

Fig. 13. Constraints defining TrainRoute L2 (not optimized)

Since the instance semantics model is representing a particular model with known associations, we implement static functions to represent these associations. This involves removing the fields and replacing them by static functions. The redefined TrainRoute signature is illustrated in Fig. 14. Static functions representing the associations in the model is illsutrated in Fig. 15. For each field, we have created a particular function with the same name as the field. Furthermore, a function defines a particular relation (i.e. TrainRoute->TrackCircuitEndpoint), and a union of all the

possible relations between these kinds of signatures. Note that Fig. 15 only represents two of the functions involving TrainRoute, but that these functions correspond to all instances of TrainRoute.

Fields are normally used by modeling the relational join operator ".", and these static functions can be used in the same manner. E.g. by using relational join on a TrainRoute and a function, representing a relation from the TrainRoute, the result will be the related object defined by the function. Thus, implementing these functions does not require big changes to other Alloy models using the signatures and fields. The only change that may be needed is according to the name-collision of functions. Since Alloy only has one namespace, function names need to be unique. Thus, if fields of different signatures have the same name, the name of the functions, representing the fields, have to be updated (i.e. names of functions trStart and trEnd in Fig. 15 are updated because of name-collision with functions representing the Track fields in Fig. 1). The TCL dynamic semantics model, which uses the signatures and fields from the static and instance semantics model, does not need to be updated if no name-collisions occur.

```
abstract sig TrainRoute {}
```

Fig. 14. A TrainRoute defined by the static semantics (optimized)

```
one sig tr_L2 extends TrainRoute{}

fun trStart[]: TrainRoute->TrackCircuitEndpoint {
      tr_BI2->ep_TCE1 +
      tr_BII1->ep_TCE1 +
      tr_L2->ep_TCE2 +
      tr_N1->ep_TCE3 +
      tr_M2->ep_TCE4 +
      tr_O1->ep_TCE5 +
      tr_AI2->ep_TCE6 +
      tr_AII1->ep_TCE6
}

fun trEnd[]: TrainRoute->TrackCircuitEndpoint {
      tr_BI2->ep_TCE4 +
      tr_BII1->ep_TCE5 +
      tr_L2->ep_StationA +
      tr_N1->ep_StationA +
      tr_M2->ep_StationC +
      tr_O1->ep_StationC +
      tr_AI2->ep_TCE2 +
      tr_AII1->ep_TCE3
}
```

Fig. 15. Static functions defining the TrainRoute relations (optimized)

The results of running the simulations presented in Section 3 with the partial instance optimization implemented are illustrated in Fig. 16 for the two-track station and in Fig. 17 for the three-track station. Note that the number of variables and clauses in are greatly reduced (i.e. the number of variables of the simulation of the three-track station with 4 trains is reduced from 1M to 681k and the number of

clauses is reduced from 3.5M to 2.1M). Also note that the analysis time is reduced from about 8 minutes to about 40 seconds.

```
Executing "Run run$1 for exactly 19 State, 2 Train, 1 Station, 8 TrainRoute, 6 TrackCircuit, 8 Track, 10
Endpoint, 6 Signal, 4 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  79573 vars. 1146 primary vars. 219504 clauses. 3078ms.
  Instance found. Predicate is consistent. 282ms.

Executing "Run run$1 for exactly 37 State, 4 Train, 1 Station, 8 TrainRoute, 6 TrackCircuit, 8 Track, 10
Endpoint, 6 Signal, 4 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  304117 vars. 3638 primary vars. 1022035 clauses. 12985ms.
  Instance found. Predicate is consistent. 12033ms.
```

Fig. 16. Analysis effort with partial instances for simulation of two-track station

```
Executing "Run run$1 for exactly 19 State, 2 Train, 1 Station, 12 TrainRoute, 7 TrackCircuit, 11 Track, 14
Endpoint, 8 Signal, 5 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  176848 vars. 1507 primary vars. 462105 clauses. 6877ms.
  Instance found. Predicate is consistent. 609ms.

Executing "Run run$1 for exactly 37 State, 4 Train, 1 Station, 12 TrainRoute, 7 TrackCircuit, 11 Track, 14
Endpoint, 8 Signal, 5 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  681795 vars. 4711 primary vars. 2100857 clauses. 31253ms.
  Instance found. Predicate is consistent. 8937ms.
```

Fig. 17. Analysis effort with partial instances for simulation of three-track station

4.3 Subsetting

The last optimization we look at in this paper is the use of subsets of signatures to hint about the number of instantiated objects. Since the analysis effort searching all possible instances and discarding illegal solutions is huge, a hint about the number of objects in the solution can reduce the effort by a large amount. E.g. if we know that a TCL model contains eight TrainRoutes, this optimization involves making eight sub-signatures of TrainRoute, where each signature only can be instantiated once (using the *one* keyword in from of the sub-signature). This is illustrated in Fig. 13 and Fig. 15, where *tr_L2* extends TrainRoute, and only one instantiation of tr_L2 is allowed. As for partial instances, subsetting requires the model to be partly fixed, and is therefore in particular applicable when analyzing a given instance model.

We have chosen to use this optimization in this paper due to the analysis effort without using it. Therefore the TCL instance semantics model extends the signatures in the static semantics model and only allows them to be instantiated once. Omitting the keyword *one* in the instance semantics model leads to the results illustrated in Fig. 18. This figure shows that the simulation performed in Section 3 with a two-track station and two trains was not practically solvable, and was stopped after two hours of generating the CNF.

```
Executing "Run run$1 for exactly 19 State, 2 Train, 1 Station, 8 TrainRoute, 6 TrackCircuit, 8 Track, 10
Endpoint, 6 Signal, 4 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  Generating CNF...

Solving Stopped.
(Stopped after 2 hours).
```

Fig. 18. Analysis effort without proper subsets for simulation of two-track station

5 Discussion

As we have seen in Section 4, optimizations of Alloy models can greatly reduce the analysis effort required. In this paper, we have focused on applying these optimizations to TCL models to illustrate how to use them. However, the optimizations are not specific to TCL, but can be applied to Alloy models in general. We have implemented these optimizations to examples in other domains with similar results. Therefore, we believe that the results presented in this paper should apply for examples in other domains.

The examples presented in this paper are used to illustrate how optimizations can lead to reduced analysis effort. They illustrate that even with relatively small models the analysis effort can be large. As we have seen, increasing the station models slightly, i.e. adding another track, greatly escalates the analysis effort required. Using a platform-specific SAT-solver can reduce the analysis time. However, the analysis time and effort will grow exponentially when the model grows. Therefore, optimizations of the Alloy models, which show huge potential, are significant for being able to analyze larger models.

In Section 4 we presented the difference in analysis effort for each optimization. We have combined the optimizations, leading to the results illustrated in Fig. 19 for the two-track station and Fig. 20 for the three-track station. Note that the number of variables and clauses are reduced even more than earlier when we applied only a single optimization.

```
Executing "Run run$1 for exactly 19 State, 2 Train, 1 Station, 8 TrainRoute, 6 TrackCircuit, 8 Track, 10
Endpoint, 6 Signal, 4 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  46171 vars. 1146 primary vars. 154182 clauses. 1922ms.
  Instance found. Predicate is consistent. 469ms.

Executing "Run run$1 for exactly 37 State, 4 Train, 1 Station, 8 TrainRoute, 6 TrackCircuit, 8 Track, 10
Endpoint, 6 Signal, 4 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  174025 vars. 3638 primary vars. 767623 clauses. 8984ms.
  Instance found. Predicate is consistent. 21889ms.
```

Fig. 19. Analysis effort with optimizations for simulation of two-track station

```
Executing "Run run$1 for exactly 19 State, 2 Train, 1 Station, 12 TrainRoute, 7 TrackCircuit, 11 Track, 14
Endpoint, 8 Signal, 5 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  68358 vars. 1507 primary vars. 248279 clauses. 3860ms.
  Instance found. Predicate is consistent. 219ms.

Executing "Run run$1 for exactly 37 State, 4 Train, 1 Station, 12 TrainRoute, 7 TrackCircuit, 11 Track, 14
Endpoint, 8 Signal, 5 Stiller, 2 Building, 0 SLock, 0 Derailer"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  259255 vars. 4711 primary vars. 1268061 clauses. 18781ms.
  Instance found. Predicate is consistent. 7469ms.
```

Fig. 20. Analysis effort with optimizations for simulation of three-track station

6 Related Work

The Alloy Analyzer performs analysis by translating the Alloy model to Kodkod formulas, which are used to search for a solution. Kodkod allows several forms of optimizations, such as symmetry breaking, partial instances etc. A comparison of Kodkod and Alloy 3 (without Kodkod) is performed by Torlak and Jackson [15], leading to the use Kodkod in Alloy 4. Torlak and Dennis [14] present Kodkod and its optimizations, including partial instances on a sudoku example. They show the difference between Alloy formulas and Kodkod formulas. However, their motivation is to explain Kodkod, while in this paper we have focused on how to apply optimizations to Alloy models and the concrete impacts on example models and simulations.

Frias and Galeotti [4] present an optimization to the analysis of DynAlloy models. DynAlloy is an extension to Alloy for adding dynamic properties (see [5]). The optimization is based on dividing the objects into two sets, mutable (objects whose state can change) and non mutable. They are thus able to reduce the scope necessary to perform analysis. As a comparison, our paper is based on standard Alloy, and present general optimizations for Alloy models.

7 Conclusion and Future Work

In this paper we have presented three optimizations to Alloy models, and shown that the benefit of optimizing Alloy models is huge. We also discussed when and how to apply the optimizations. Calculating all possible solutions and discarding the illegal ones is quite expensive. Thus, reducing the complexity of expressions or specifying directly the structure of the model being analyzed, through proper subsets or partial instances, can lead to a large performance gain. This is especially important when the models being analyzed grow in size, since then even changing to a more efficient SAT-solver will not be beneficial enough. The optimizations were illustrated on concrete TCL example models, showing the performance gains of performing the simulations on these models.

The optimizations have been applied to and evaluated by several TCL models and models of other domains. Further study of the optimizations using larger models in other domains is important future work. Other significant future work involves analysis of Alloy models to automate the process of finding and suggesting how and where to apply the optimizations.

Acknowledgements. The work presented here has been developed within the MoSiS project ITEA 2 – ip06035 and the Verde project ITEA 2 – ip8020 parts of the Eureka framework.

References

1. Alloy4. http://alloy.mit.edu/alloy4/
2. Eén, N. and Sörensson, N.: An Extensible Sat-Solver. In, E. Giunchiglia and A. Tacchella, (eds.) Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science, vol. 2919, pp. 333-336. Springer Berlin / Heidelberg (2004)
3. Endresen, J., Carlson, E., Moen, T., Alme, K.-J., Haugen, Ø., Olsen, G.K., and Svendsen, A., "Train Control Language - Teaching Computers Interlocking", Computers in Railways XI (COMPRAIL 2008), Toledo, Spain, (2008)
4. Frias, M.F. and Galeotti, J.P., "Faster Sat-Based Analysis of Oo-Programs by Separation of Mutant and Non Mutant Objects", in First ACM Alloy Workshop. Portland, Oregon, USA, (2006)
5. Frias, M.F., Galeotti, J.P., Pombo, C.G.L., and Aguirre, N.M., "Dynalloy: Upgrading Alloy with Actions", in 27th International Conference on Software Engineering (ICSE '05). St. Louis, MO, USA: ACM, (2005)
6. Gomes, C.P., Kautz, H., Sabharwal, A., and Selman, B.: Satisfiability Solvers. In: Handbook of Knowledge Representation. Elsevier (2007)
7. Kelly, S. and Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, Inc., (2008)
8. Kelsen, P. and Ma, Q., "A Lightweight Approach for Defining the Formal Semantics of a Modeling Language", in Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems. Toulouse, France: Springer-Verlag, (2008)
9. Kodkod. http://alloy.mit.edu/kodkod/
10. MiniSat. http://minisat.se/
11. SAT4J. http://www.sat4j.org/
12. Svendsen, A., Møller-Pedersen, B., Haugen, Ø., Endresen, J., and Carlson, E., "Formalizing Train Control Language: Automating Analysis of Train Stations", in Comprail 2010, B. Ning and C. A. Brebbia, Eds. Beijing, China: WIT Press, (2010), pp. 245-256
13. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.-J., and Haugen, O., "The Future of Train Signaling", Model Driven Engineering Languages and Systems (MoDELS 2008), Tolouse, France, (2008)
14. Torlak, E. and Dennis, G., "Kodkod for Alloy Users", in First ACM Alloy Workshop. Portland, Oregon, USA, (2006)
15. Torlak, E. and Jackson, D.: Kodkod: A Relational Model Finder. In, O. Grumberg and M. Huth, (eds.) Tools and Algorithms for the Construction and Analysis of Systems, vol. 4424, pp. 632-647. Springer Berlin Heidelberg (2007)