4. Resource Distribution Model

4.1 Requirements

In this section we will investigate the requirements for the RDM. We will achieve this by performing an commonality/variability analysis [**?**]. This will reveal what the common features are on which we may depend and the variation which we will need to account for.

4.1.1 Commonality/variablity analysis

Definitions

Resource: Any measurable/calculable parameter of a system

Resource constraint: A constraint imposed on a resource.

Component: Any physical or hypothetical entity that can consume or produce a resource

Quality of Service (QoS): Parameters which are indicative of the level of service of a system.

Commonalities

C1.1 A resource can be consumed or offered by multiple components.

C1.2 A component can produce or offer multiple resources.

C1.3 Resources are scarce, i.e. the amount produced must exceed the amount consumed.

C1.4 Resources are correlated and can be converted into one another.

C1.5 Resource amounts can be used to objectively compare functionality of a system.

Variabilities

V1.1 Though all use cases agree on the above commonalities, we cannot predict all resources, components, constraints and interconnection that can occur.

V1.2 Resources of a system can be modelled on a micro-scale or macro-scale.

- A micro-scale (e.g. a single sensor) entails concrete, palpable parameters.

- A macro-scale (e.g. an entire WSN application) entails accumulated, theoretical parameters

V1.3 A system can have multiple resources as QoS indicators

V1.4 Short term resource usage (e.g. interval of seconds) requires a different granularity than long term resource usage (e.g. interval of days).

V1.5 Some resources are directly measurable and thus known for a certain moment of measurement. However, some resources are derived and calculated using other resource values. [**?**]

V1.6 Most resource values differ depending on system's measured state

V1.7 Some resource values/usages differ depending on a specific system function

V1.8 Given a system's state some system functions are better suited than others.

4.1.2 Requirements

R1.1 The model should represent resource distribution in a system

R1.2 Resources should be able to be transformed into other resources (many-to-many)

R1.3 The model should account for the fact that the value of a resource can originate from different sources. The identified sources are the following:

constant a predefined value specified on development time (e.g. initial battery capacity),

measured a value specified as observed on run time (e.g. percentage of battery capacity left),

calculated derived from measured values (e.g. runtime left),

variable any value or a calculation depending on specific system function (e.g. power usage).

R1.4 Each model should have one, and only one, resource that is associated with a heuristic QoS function.

R1.5 A model should contain constraints that describe the limitations of interconnected resources.

R1.6 Given a resource distribution model, constant-valued resources and measurements, for each combination of values for variable resources, a value should be able to be evaluated for each calculated resource

R1.7 Given a calculable resource distribution model (R1.6), a set of resource constraints and an optimizer function; an optimal, valid appointment for each variable resource value should be able to be solved efficiently.

4.1.3 Justification

Table 1.1 demonstrates how the proposed requirements account for the determined variety, based on the observed commonalities. Most requirements can easily be traced to the variety it strives to restrain. An exception is requirement R1.4, which states that one resource is used to optimize the QoS. This is seemingly contradicted by V1.3 which states that multiple resources can be

Variety	Requirements
V1.1	R1.1, R1.3, R1.5
V1.2	R1.1, R1.3
V1.3	R1.2, R1.4
V1.5	R1.2, R1.3
V1.6	R1.3
V1.7	R1.3, R1.6
V1.8	R1.4, R1.5, R1.7

Requirement	Commonalities
R1.1	C1.1, C1.2
R1.2	C1.4
R1.3	
R1.4	C1.4, C1.5
R1.5	C1.3
R1.6	C1.4
R1.7	C1.3, C1.5

Table 4.1: Justification of requirements by variety and commonalities

indicative of the level of QoS. This is however explained with use of C1.4. This commonality states that resources can be transformed into one another (many-to-many). It can therefore be inferred that it is possible to transform multiple QoS markers into a single optimizable, meta-physical resource, according to some heuristic QoS function.

Evidently omitted from the justification table is variation V1.4. This is due to that a this variety has far-reaching consequences for the implementation of the model. Therefore a choice has been made to focus on modelling of resource distribution during large time intervals. This choice will elaborated in section 1.3.2.

4.2 State of the art

Work regarding modelling resource distribution has been performed in several studies. Elementary examples of such research are the studies of Ammar et al[**?**]. Through their efforts they laid the ground work for representing entities interconnected by shared resources. This UML-based model was one of the first examples of such a representation using formal methods and tools. Another example of early research is the study performed by Seceleanu et al[**?**]. This study focussed on modelling resource utilization in embedded systems using timed state machines. The transitions in these automata were attributed resource costs to model the consumption of resources for Transitioning to a state of remaining in one. Resource consumption and performance over time can then be calculated and analysed according to the paths taken in this model.

A continuation of this work was performed by Malakuti et al[**?**]. They combined the methods of the previous authors by provisioning the modelled system components with their own state machines. These state machines model the resources and services that are offered and required by the components. By analysing these component models as composite state machines, model checking tools (such as UPAAL[**?**]) can be used to analyse and evaluate the performance of the investigated system as a whole.

4.3 Solution

4.3.1 Solution options

These efforts have produced methods of representing components connected by shared resources. Especially the notation of Malakuti et al[**?**], which is both intuitive and descriptive. We will therefore continue to use this notation.

however these models are all focussed on components that are self-aware of their resource usage and performance. Instead, we are interested in off-site analysis of interconnected resources and accumulated performance of a composite system. Our focus is therefore alternatively more resource-centred. It is concerned how production and consumption of a resource is interconnected. Components only serve as secondary elements, merely specifying how these resources are converted into other resources. Therefore a resource-centred adaptation of this framework might be more suitable for our problem.

Secondly, there is the issue of how to represent a Resource Utilization Models (RUM)[**?**], the model for variable behaviour of components. Previous studies [**?**, **?**] have used timed automata to represent behaviour cycles. This allows for automated tools to calculate a runtime schedule in high levels of granularity. However the high level of granularity comes at the cost of efficiency. When we shorten the time intervals for the automata, entailing higher granularity, then solvers require additional computational resources and time to execute. This might force a problem on resource constraint devices or applications that require the solver algorithm to run many times for a multitude of devices. Additionally, we need to consider that a model contains multiple components specified by RUM's. For these models a valid, optimal RUM composition needs to be determined. In this case RUM's might influence each other, which implies that for different compositions of these models, the individual models need to be re-calculated.

An alternative approach is to model the RUM as a set of static parameters. A component then has multiple RUM's representing different modes of execution. This is achieved by averaging the behaviour for that mode of execution, which would otherwise be modelled by a single timed automaton. This comes at great cost of granularity, since the RUM's now only describe a few static, predefined long-term behaviours. However it significantly improves the complexity of the search space. For this approach timed automata is no longer a sensible technology since the element of time intervals has been eliminated. Instead the problem is a pure decision problem[**?**]. The only problem to be solved is to find a suitable RUM for each modelled component. The search space of a decision problem can be explored with a simple brute force search, exploring all options and compositions. However more effectively, combinatorial problems can often be solved with constraint solvers. The problem is easily transposed to a constraint problem with the RDM as model, resource constraints as constraints and the RUM's as variables for the components. With the many solution strategies described in **??** available for different types of problems, a suitable solver should be able to be found or developed.

4.3.2 Solution choices

With careful consideration the following choices for the solution implementation have been made. For modelling we chose to adapt the framework of Malakuti et al[**?**], by emphasizing on resources and introducing some new features. The components will still exist in the model, but will merely serve the function of connecting two resources to one another. Another adaptation is the existence of multiple RUM's for a component, which allows injection of different methods of operation and calculation of the optimal system functionality.

As for how to model the RUM, we chose to reduce the complexity of the system by modelling variable resource usage with static parameters. The strongest advocate for this choice is the fact of the focus for this research: large IoT applications. In an IoT monitoring platform the task of determining optimal device function will need to be performed repeatedly for many sensor devices. Additionally, devices in most large scale IoT applications only send and receive data a few times per day[**?**]. Therefore high granularity is not of grave importance because the feedback-control cycle is not that short.

The fact that a component can have more than one mode of operation and the choice of static parameters for those functions, makes constraint solvers most suitable as means to solve the model. We will however adapt the search algorithm to conclude not only the valid compositions but the optimal solution, given some heuristic function.

4.4 Design

4.4.1 Model

As stated we will model resource distribution by extending the model by Malakuti et al[**?**]. The chief adaptations in our model are:

1. the inclusion of a single explicitly defined optimised resource,
2. RUM‘s with static resource values,
3. the existence of multiple RUM‘s for a single component, and
4. constraints defining valid resource interconnectivity:
   (a) implicit constraints enforcing availability: $R_{offered} \geq R_{consumed}$
   (b) additional explicit constraints specified by developer

A graphic representation of the adapted meta-model can be found in figure 1.1. A complete entity relation diagram for the meta-model can be found in Appendix **??**. To illustrate the application of this meta-model, an example of an instantiation of the model can be found in 1.2.

In essence the model is a collection of *Resources* and *Components*. Each of these resources can be connected to components by means of a *ResourceInterface* and a *ResourceFunction*.

Resource

A resource is an entity describing a parameter of a system. This can be a measured parameter (e.g. battery capacity or throughput), but can also describe

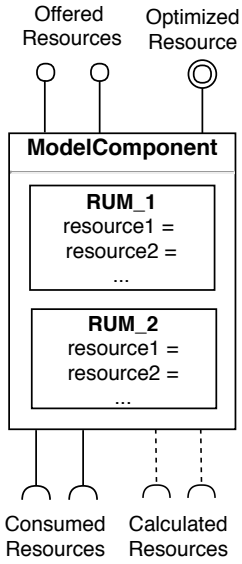


Figure 4.1: Notation of an RDM component with RUM's

a derived parameter (e.g. service time left). Each resource is identified by it's name and has a unit associated with it. By aggegating the ResourceInterfaces of a resource the amount of the resource produced and consumed can be collected and analysed.

ResourceInterface

Resources and components are connected through resource interfaces. A ResourceInterface can be one of three types:

Offer Indicating that the component produces an amount of the resource,

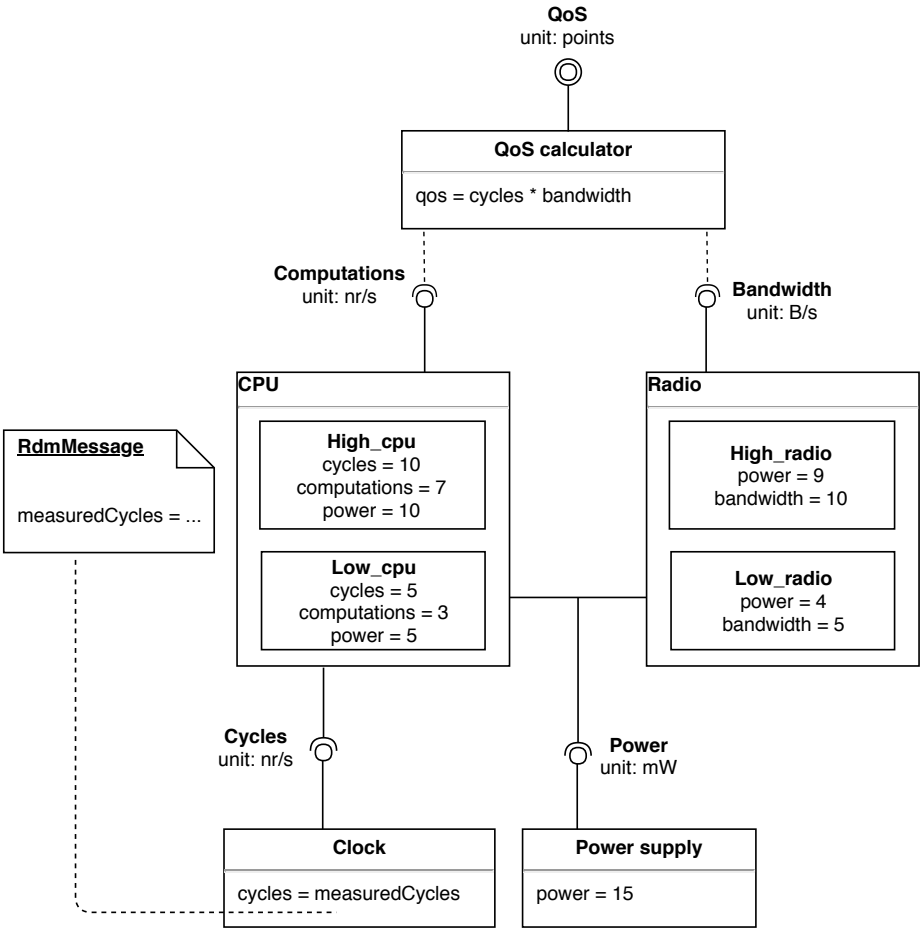
Consume Indicating that the component consumes an amount of the resource,

Calculate Special consume relation. This interface supplies 100% of the offered resource, without formally consuming any amount. This relation is used to further calculate with the offered value, without it impacting the constraints of the resource. For example a QoS indicator that is "consumed" by a general QoS calculation.

Each interface has a value specifying the amount of the resource produced or consumed by the component. This value is repeatedly set and evaluated at runtime by executing a ResourceFunction.

Component

Any entity producing, consuming and converting a resource is represented by a component. A component can therefore be a physical entity such as a radio module or a battery or a hypothetical entity such as a QoS calculator executing a heuristic function. A component possesses a ResourceFunction of each Resource it is connected to.



Constraints:
$c_1 : cycles_{clock} >= cycles_{CPU}$
$c_2 : power_{power\_source} >= power_{CPU} + power_{Radio}$

Optimize:
$max(QoS)$

Figure 4.2: Example instatiation of the RDM meta-model with a CPU and a radio

A special case of the Component is the ModelComponent. This class inherits all functionality of the ordinary Component, but its ResourceFunctions are extracted from one of its RUM's. Each RUM describes the parameters during one mode of operation of the components. This allows runtime analysis of variable behaviour as effect of different functionalities.

ResourceFunction

The value of a ResourceInterface is determined by a ResourceFunction. It consists of a function that takes a double array as argument and has a double as result, and an array of resource identifiers. Runtime solvers or engines will then fill the input array according to the resource identifiers in order to execute the function. ResourseInterfaces can be compactly instantiated using lambda expressions and VarArgs. E.g.:

```
1 ResourceFunction totalServiceTime = new ResourceFunction(
2     (x)->x[0]+x[1], "yearsServed", "yearsLeft"
3 );
```

To model the intended behaviour of the model we introduce a set of *Requirements* and an *Optimizer*.

Requirement

A resource can have a number of Requirements as constraints that limit the possible values of variation for that resource. The standard built-in requirement for every resource is the *OfferConsumeGTE* requirement which enforces that the amount produced needs to be greater or equal than the amount consumed. Additional requirements *OfferConsumeEQ* and *RangeRequirement* are specified, that respectively require the exact amount offered to be consumed and the amount offered or consumed to be within certain bounds. Finally the abstract class *Requirement* can be extended by a developer to specify any tailored requirement.

Optimizer

To ascertain the heuristic score of an RDM with an injected RUM configuration we introduce the Optimizer. The Optimizer is an extended class of Resource of which exactly one must exist in an RDM. The optimizer takes the evaluated offered amount of this resource and calculates a score. This score is a value on a comparative scale on which a higher value implies a more optimal solution. Specified are the *MinMaxOptimizer* which evaluates that the amount offered must have a minimal or maximal value and the *ApproxOptimizer* which evaluates that the resource must have an amount offered as close to a specified value as possible. However, custom implementations of the Optimizer can again be made by developers.

RdmMessage

Finally, to supply the model with the state of the system under investigation, we pose the RdmMessage. The RdmMessage is provisioned using values measured from the system and injected into the model, after which the appropriate

resource values are evaluated accordingly. Technically, a simple mapping from a resource identifier to a measured value would suffice for this purpose, but this mapping is wrapped in an object to support future evolution.

4.4.2 Solving the model

With the model well established we can now try and solve the model. From requirement R1.7 we find the goal of solving the model is to find a composition of RUM's such that:

1. each ModelComponent has exactly one RUM associated with it,
2. all resource constraints are satisfied, and
3. the optimizer function of the optimized resource has the highest score.

The first and second requirement imply constraint solvers as an applicable technology, since they are effective in finding a valid solution for a constraint decision problem. However, the third requirement entails that we do not want to find just any valid solution, but the *optimal* valid solution. In order to do that we need to consider every valid solution to the problem and compare how they compare heuristicly. This entails a full brute force search approach through the entire search space of RUM compositions. We can however use constraint solver paradigms to preventively reduce the search space as we search through it.

The way we do this is by employing backtrack search. In a simple brute force search we would calculate all RUM compositions (Cartesian product) and for each composition we provision the full model and evaluate it. Instead we will iteratively select a component and one of its models. We will then not provision the entire model, but inject only the selected model in the chosen component. Consequently, we set the values for variables for which we can resolve a definite value, given the current state of the model. We then evaluate the resource constraints. Given an incomplete model any constraint can have one of three statuses:

- satisfaction,
- failure, or
- uncertain

for all consequent assignments of unprovisioned components.

If a constraint evaluates to *satisfied* it will be pruned from the constraint set and will not evaluated for the remainder of this branch of the search tree, since we know it will always succeed. If a constraint is *uncertain* we keep it, since we do not know its status for each and every future state. If even a single constraint *fails* we know the remainder of this branch of the search tree will never be valid. Therefore we backtrack through the tree by partially rolling back model assignments. We then select a different model for the same component or a different component entirely and repeat the algorithm. This way we do not recheck constraints we already know the state of and do not evaluate paths we know will not satisfy the constraints. The full original algorithm is given in Listing 1.1.

Given that we encounter unsatisfactory options early in the tree, this will possibly eliminate large parts of the search tree. An example of the application of this algorithm on the example previously posed (Figure 1.2) is given in

Figure 1.3. This application illustrates that using this algorithm, we eliminate a significant portion of the search tree. This is due to early constraint failure detection in the *CPU=high_cpu* banch of the tree.

Backtracking
Input: A constraint network R and an ordering of the variables $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

1. (Initialize.) $cur \leftarrow 0$.
2. (Step forward.) If x_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, $cur \leftarrow cur + 1$. Set $D'_{cur} \leftarrow D_{cur}$.
3. (Choose a value.) Select a value $a \in D'_{cur}$ that is consistent with all previously instantiated variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$ (x_{cur} is a dead-end), go to Step 3.
   (b) Select a from D'_{cur} and remove it from D'_{cur}.
   (c) For each constraint defined on x_1 through x_{cur} test whether it is violated by $\overrightarrow{a}_{cur-1}$ and $x_{cur} = a$. If it is, go to Step 2a.
   (d) Instantiate $x_{cur} \leftarrow a$ and go to Step 1.
4. (Backtrack step.) If x_{cur} is the first variable, exit with "inconsistent". Otherwise,set $cur \leftarrow cur - 1$. Go to Step 2

Listing 4.1: Algorithm for backtrack search[**?**]

4.5 Discussion

Static model

As stated before we chose to use a static representation of resource utilization by ModelComponents. We chose this in order greatly reduce the complexity of the problem and this allows the model to be evaluated within a reasonable amount of time. We came to this conclusion after early experiments with timed automata. In this experiment we modelled a minimal system with one component with three RUM's. When analysing the model using time intervals of one week over a life span of ten years, it took over one minute to calculate the optimal traversal of the automaton. Granted, this was performed on a laptop machine and not a high-powered server. When deployed on a server with sufficient calculatory resources the time to calculate will be reduced. This is however counteracted by the fact for a WSN application this calculation needs to be repeated for thousands of sensors. When we compare this performance to that of the static models, which can evaluate more complex models (e.g. 3 components, 5 RUM's each) within seconds, we must eliminate timed automata as valuable real-time technology. However, this does not eliminate automata entirely. Automata can still be used to model the fine grained run cycles of parts of a system in order to develop generalized static RUM's.

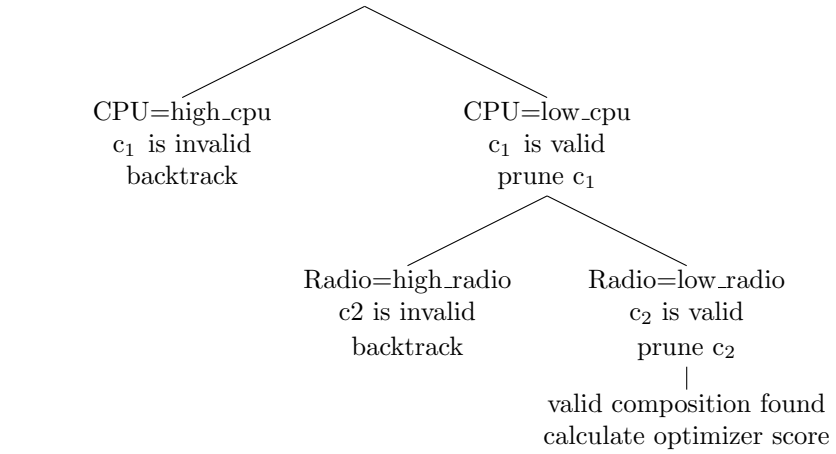


Figure 4.3: Application of backtrack search on RDM of Figure 1.2

Solver libraries

When developing this solution we chose to implement the constraint solving algorithm ourselves, instead of employing existing libraries such as Choco Solver[**?**] or OptaPlanner[**?**].

The Choco Solver is a powerful solver which not only employs backtrack search, but also constraint propagation to eliminate failing search paths before assigning them. However, while powerfull, it has only limited support for real intervals [**?**]. Additionally it proved very difficult to convert the user defined models and arithmetic expressions to the modelling mechanism of the solver. Requiring the user to either input the model and calculations in the complex modelling mechanism of the Choco Solver or for us to develop a compiler to rewrite the easy to write user input to Choco Solver code.

Another examined library is the OptaPlanner. The OptaPlanner is a modelling framework for constraint problems and excels in use cases involving planning and resource allocation. It also enables object injection which would be greatly suitable for injecting our RUM's into components. However the OptaPlanner is strictly a constraint modelling framework and does not employ advanced solving techniques developed in the field of constraint programming. It performs a brute force depth-first search over the search space (Cartesian product of all RUM compositions) running a single code block which evaluates all constraints. It consequently can not reduce the search space by eliminating failing branches and redundant constraints. Therefore it lacks the means to solve the problem efficiently

Finally, the implementation of backtrack search does not differ much from the implementation of depth-first search. Additionally, developing our own solver allows us to incorporate domain knowledge into our custom search al-

gorithm, further reducing the runtime required. This reduces the comparative benefit of employing a constraint solver library and eventually led us to develop our own solver implementation.

Constraint propagation

A technique in constraint solvers mentioned before is the concept of constraint propagation. Constraint propagation explores the search space the in the same manner as backtrack search. However, for each variable assignment V_1 all other variable domains are preventatively reduced by pruning all variable assignments V_2 that are incompatible with V_1. For example in the example of Figure 1.2: if *CPU=High_CPU* is initially assigned, *Radio=High_radio* is pruned because it would require more power than is actually produced. This eliminates inconsistent variables without the need of assigning them, thereby reducing the search space even more effectively than native backtrack search. This is easily implemented with integer/real variables that are interconnected with constraints. However, in our model the variables are not integer/real domains, but objects with integer/real variables. This doesn't make constraint propagation impossible, but does complicated it greatly.

Secondly, the interconnected nature of our problem can impede the benefits received from constraint propagation. To illustrate this consider the following example: resource R is connected to a set of producers P and a set of consumers C, for each the amount produced or consumed is variable. The amount produced or consumed by any component x is denoted by R_x. The availability constraint (more must be produced than is consumed) on R can then be written as:

$$\sum_{p \in P} R_p \geq \sum_{c \in C} R_c$$

Which entails for any consumer $c1 \in C$:

$$R_{c1} \leq \left(\sum_{p \in P} R_p - \sum_{c2 \in (C - c1)} R_{c2} \right)$$

In order to be able to prune any value from the domain of consumer *c1*, we need to assign all producers in order to determine a reliable upper bound[1]. This requires the search to be already at least $|P|$ levels deep, reducing the part of the tree possibly eliminated. Even then, we are only able to prune the values for which:

$$R_{c1} > \sum_{p \in P} R_p$$

Which might not be many since a single consumer must consume more of a resource than produced by all producers combined, in order for the constraint to fail. When other consumers get a value assigned we may be able to prune values more easily, but this requires even more variable assignments. This problem is aggravated when R_p is a derived value calculated using a number of other resources. Values for all these resources must be known in order to calculate the value of R_p.

[1] Future assignments of the other consumers may be disregarded since they will never raise the upper bound for R_{c1}, only lower it.

To conclude, the part of the tree that is eliminated with constraint propagation is limited since we are already halfway into the search tree and, additionally, the chance that a value is eliminated halfway in the tree is very small. Therefore no further effort was made to incorporate constraint propagation or other look-ahead strategies in the solver.