Incremental Runtime-generation of Optimisation Problems using RAG-controlled Rewriting

René Schöne*, Sebastian Götz*, Uwe Aßmann* and Christoff Bürger†
* Software Technology Group, Technische Universität Dresden
(rene.schoene|sebastian.goetz1|uwe.assmann)@tu-dresden.de
† Department of Computer Science, Faculty of Engineering, Lund University, Sweden
christoff.burger@cs.lth.se

Abstract—**In the era of Internet of Things, software systems need to interact with many physical entities and cope with new requirements at runtime. Self-adaptive systems aim to tackle those challenges, often representing their context with a runtime model enabling better reasoning capabilities. However, those models quickly grow in size and need to be updated frequently with small changes due to a high number of physical entities changing constantly. This situation threatens the efficacy of analyses on such models, as they lack an efficient management of those changes leading to unnecessary computation overhead. We propose applying scalable, incremental change management of runtime models in the presence of a complex model to text transformation. In this paper, we present and evaluate an example of code generation of integer linear programs. In our case study using synthesized models, we saved 35 - 83% processing time compared to a non-incremental approach. Using our approach, future self-adaptive systems can handle and analyze large-scale runtime models, even if they change frequently.**

I. INTRODUCTION

A major challenge regarding the integration of computer-based systems with the physical world is the development of convenient runtime models [1], [2]. Runtime states of cyber-physical systems [3] have to be represented in such a way, that they can be efficiently updated and analyzed for self-adaptation. This comprises monitoring of change-events, analysis of the resulting system state, planning of reactions and the actual execution of them – a feedback loop [2].

A particularly challenging part of such feedback loops is the deduction of reactions to updates, i.e., the reasoning in the analyze and plan phase [4]. Often, logical or mathematical formalisms are used to ensure only efficient and correct plans are derived, e.g., integer linear programming (ILP) for optimizing self-adaptations [5] or logic-based structural reasoning like Alloy [6]. Although such techniques ease the deduction of reaction-plans, they still require the derivation of an actual problem specification suitable for their respective reasoning tooling. Since changes in runtime models typically happen frequently and mostly affect only small model parts [7], planning problems should be deduced by *incremental* analyses. Given a model update, analyses should only be re-evaluated if influenced by the update. If not, previously cached results should be reused. The benefit of such an incremental derivation of planning problems increases with model size. Hence, in the scope of the *Internet of Things*, i.e., in the presence of very large runtime models, it is of uttermost importance [8]. A common solution is to manually implement a cache mechanism. However, this leads to other serious problems like inconsistencies [9].

Building on reference attribute grammar controlled rewriting – a promising technology to accomplish automatic, incremental analyses of runtime models [10] – we will show its application in our case to generate ILP optimisation problems. Reference attribute grammars (RAGs) [11], are a declarative formalism to specify semantics for tree structures. A RAG extends tree instances of a context-free grammar to abstract syntax graphs by superimposing a semantic overlay graph. In contrast to ordinary attribute grammars [12], RAGs do not support incremental evaluation out-of-the-box. For that reason, we use RAG-controlled rewriting [10], which permits model updates in terms of graph rewrites that in turn update analyses, thus automatically achieving incremental evaluation. Using RAG-controlled rewriting, we are able to model software systems including their context, incrementally reason on those systems and still cope with runtime model changes. In this paper, we approach the following research questions:

RQ1 Are reference attribute grammars a suitable modeling framework for large-scale runtime models?
RQ2 Is it possible to incrementally propagate changes in the runtime model to domain-specific code generated from this model?
RQ3 If yes, how beneficial is incremental change propagation compared to non-incremental solutions?

In the following Section II, we describe our previous research building the foundation of this work. Subsequently, concepts and advances of our approach are described in Section III, followed by an extensive evaluation in Section IV. We compare our approach to related work in Section V and conclude in Section VI.

II. PROBLEM DOMAIN: MULTI-QUALITY AUTO-TUNING

This section describes the domain of our case study and its associated evaluation: Multi-Quality Auto-Tuning (MQuAT). We selected this problem domain because it is a complex problem involving runtime models and reasoning on them. Furthermore, our previous research [5] revealed, that incremental runtime model reasoning is a major challenge and can not be easily integrated into existing systems.

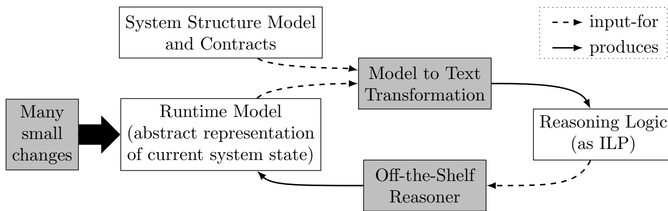


Figure 1: Overview of Multi-Quality Auto-Tuning.

MQuAT is an approach for model-driven, component-based development of self-adaptive systems. The principal idea and key contribution is to use models and a domain-specific language enabling the developer to describe the self-adaptive system under construction and generate the reasoning logic for the analyze and plan phases from these models. Consequently, the developer is enabled to focus on the problem specification instead of the technical realization, which is automated by code generation at runtime. Hence, a representation of the system's runtime state is required – a runtime model. In other words, MQuAT adheres to the models@run.time paradigm [13], which promotes the use of modeling techniques at runtime. Figure 1 depicts the general principle of MQuAT.

In [5], the generation of reasoning logic to alternative optimization languages was investigated. Here, we focus on the generation of an integer linear program (ILP) [14] as reasoning logic. In general, the optimization problem to be solved by MQuAT is a selection and mapping problem. MQuAT models systems as sets of soft- and hardware components. Each software component can have multiple implementations, each with different modes. For example, the software component *Compress* has implementations like `zip`, `pigz` and `nanozip`, to name but a few. The `nanozip` implementation can be used in different modes, which, e.g., specify the amount of preallocated memory used for compression. Hardware components typically describe a hierarchical structure, e.g., a component *server rack* comprising several *servers*, each containing *memory*, *CPU*s and *network device*s. Both, soft- and hardware components can be further detailed by specifying their properties. For example, the hardware component *CPU* has the property *frequency* and the software component *Compress* has a property *compression-ratio*. Dependencies between components are specified using contracts, that describe a provision/requirement relation on properties of two interdependent components. For example, to guarantee a maximum processing time of a compression algorithm, a minimum amount of memory is required.

To process a user request, a system described using the aforementioned concepts contains two dimensions of variability: (1) each component, which is required to process the request, has several implementations and (2) each implementation can be used in different modes on different hardware resources. Thus, the optimization problem denotes the selection of the best implementations and their best mapping to the available resources for a given user request. To solve this problem, it is transformed to an ILP as optimization language.

The generated ILP comprises three types of constraints: (a) structural constraints describing which software components are required due to dependencies between them, (b) constraints covering the variants of software-hardware mappings as a dynamic knapsack problem [15] and (c) constraints covering hardware resources, e.g., a maximum frequency of a CPU. On a more abstract level, the ILP is an equation system $Ax = b$ comprising a matrix of coefficients A, a vector of bounds b and the target variables x. Here A and b are the mentioned constraints, in x are Boolean variables representing the choice of one mode being deployed on a certain hardware component.

The two key problems of MQuAT and many other model-driven approaches for self-adaptive software are **(1) the scalability** of both the runtime model and the reasoning upon it and **(2) efficient change management**, i.e., minor changes to the runtime model should avoid a complete re-generation and -evaluation of the reasoning logic. To realize the interconnection between development models (e.g., models describing the software architecture) with the runtime model, MQuAT uses the Eclipse Modeling Framework (EMF) for both. There are alternative, more lightweight modeling framework like the Kevoree Modeling Framework (KMF) [9] as EMF is not intended to be used at runtime. In this paper, we propose the application of reference attribute grammar controlled rewriting (RAG-controlled rewriting) as alternative modeling framework, which in contrast to KMF offers incremental evaluation in addition to being more lightweight. In particular, we will show how the reasoning logic can incrementally be generated as an ILP using RAG-controlled rewriting. Therefore, our approach is situated in the analyze and plan phases of the feedback loop, as the optimization problem determines whether and how the current configuration has to be changed.

III. RAG-controlled, Incremental ILP-Generation

Attribute grammars (AGs) are a well known technique for semantic analysis in the field of compiler construction [12]. AGs use a context-free grammar to specify the structure of tree-shaped data they operate on and attributes to analyse this data. Attributes can have dependencies to one another and on tree information, resulting in a statically known dependency graph exploited by the attribute evaluator. Based on this graph, incremental evaluation can be realized [16].

Reference AGs (RAGs) are an extension, where attributes may resolve to remote tree nodes such that an semantic overlay graph is superimposed on the spanning tree [11]. This enables new scenarios, e.g., the specification of semantics in metamodels like EMF Ecore [17]. However, dependencies of reference attributes cannot be statically decided, impeding incremental evaluation. RAG-controlled rewriting overcomes this limitation by tracking dependencies dynamically such that rewrites can incorporate them for invalidation [10]. It enables dynamic incremental attribute evaluation for RAGs. The *RACR* library [10], used in our case study to implement a runtime model for MQuAT, is a reference implementation of RAG-controlled rewriting. It can be found at https://github.com/christoff-buerger/racr.

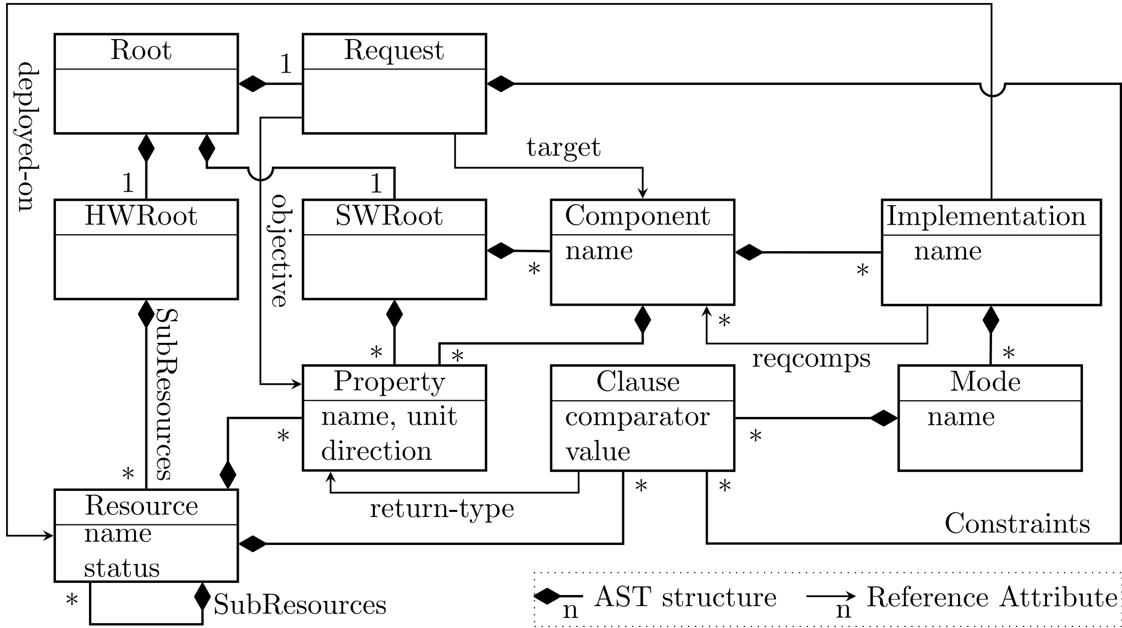


Figure 2: Grammar overview. HWRoot and Resource are hardware elements; SWRoot, Component, Implementation and Mode are software specific. The non-terminals Property and Clause are cross-cutting, whereas the Request represents the expectations from a user and triggers an evaluation of the model.

A. General Solution Overview

Figure 2 shows a simplified grammar using our approach to describe a system subject to auto-tuning, i.e., possible types of model elements and their relationships. The notation follows JastEMF [17], showing nonterminals as rectangles with terminals of them listed beneath the name of its enclosing nonterminal. Containment relations model nonterminals in production right-hands, thus, implying non-cyclic entity relations, i.e., subtrees. Arrows denote attributes which are known to be of a type the arrow is pointing to. The following three concerns are represented as nonterminals:

The hardware subtree with root `HWRoot` denotes an arbitrarily deep hierarchy of `Resources`, which are grouped by common types (e.g., CPU). Resources have a set of `Clauses` describing their current state, e.g., "CPU frequency is 1.4GHz". Clauses are linked to `Properties`, e.g., representing CPU frequency.

Software is represented as a hierarchy, too. The semantics of `Components`, `Implementations` and `Modes` are as described in Section II. Furthermore, implementations can require components, forming a dependency tree. Modes are described by a *quality contract* – a set of `Clauses` describing their non-functional behavior. With such clauses, properties like an execution time t can be described depending on input parameters, e.g., $t = n^2$, where n denotes input file size. Different modes of the same implementation can have either the same clauses, the same properties, but different values or completely different clauses.

Finally, a `Request` denotes the requested functionality along with expected values for non-functional properties. An example could be a user requesting to compress a video file, requiring a maximum execution time to finish this operation and a maximum file size after compression.

Figure 3 shows a simplified structure of an example system using the described grammar to build an AST representing such a system. It includes three parts: a) two hardware components, b) nine software components, implementations and modes (indicated with C, I and M, respectively) and c) the request. With such an AST, the structure and properties of hard- and software can be described. Further, their relationship is modeled with clauses defining requirements and provisions.

All the aforementioned elements can be modeled using RAGs. Hence, they are a suitable modeling framework for runtime models, which partially answers **RQ1**. To show that RAGs are suitable for large-scale runtime models, we have evaluated our approach in Section IV.

B. Attribution used for ILP-Generation

To realize ILP-Generation using RAGs, attributes implementing a model-to-text transformation have to be specified for each nonterminal of the grammar. The computation of these attributes is local to this nonterminal and can be reused by other attributes, as will be described later. As they are computed incrementally, the whole model transformation is incremental, thus tackling the problems revealed in Section I.

Figure 4 shows the AST introduced above in Figure 3 at runtime. It includes the most important attributes involved in the computation, their dependencies and whether they will be evaluated in a certain situation described below. The syntax of attributes will be introduced in the next paragraph. An evaluation always begins at the attribute *to-ilp*, starting the

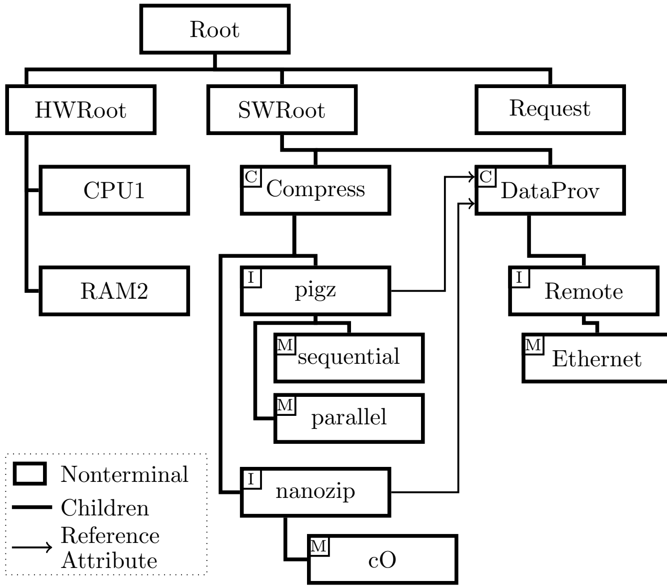


Figure 3: Example AST, built using the grammar in Figure 2.

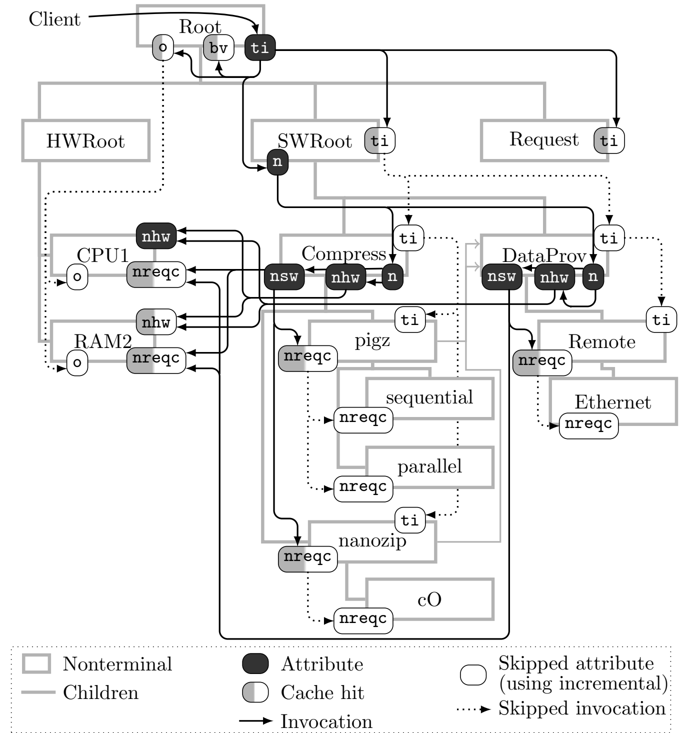


Figure 4: Attribute computation and dependencies for an example AST.

generation process. Any other invocation is controlled by *RACR*, ensuring that only those attributes are re-evaluated, whose values might have changed because they depend on updated model information. Attributes are defined to fit naturally in the structure of the ILP. In the following, the most important attributes are briefly explained. Their syntax is denoted by underlines, e.g., to-ilp is represented as `ti` in Figure 4.

objective This attribute is defined on the nonterminals Root and Resource, to compute the objective function globally (Root) or partly for the given Resource.

to-ilp Gathers the ILP constraints for each clause of the current request. Both, software and architectural constraints result from the invocation of *to-ilp* on components and implementations. Software constraints represent the dependencies of implementations and components, whereas architectural constraints ensure that only one mode of an implementation per component is deployed.

nego Models the dynamic knapsack problem (negotiation), which was outlined in Section II. Its constraints comprise property values required by components and provided by components and resources. Thus, *nego* depends on *nhw* (negotiation of hardware resources), *nsw* (negotiation of software modes) and *nreqc* (request constraints).

binary-vars Computes all used variables and their bounds.

The attribute *binary-vars* is a good example to demonstrate the strength of incremental evaluation on a small scale. If, for example, the value of a hardware entity has changed (e.g., the CPU frequency has been changed), this attribute does not need to be re-evaluated, because the change will not introduce new variables. Thus, its value will be read from cache, as all information it depends on, whether tree-structure or attribute values, are unchanged.

Figure 4 shows, which attributes need to be re-evaluated in case of a property change of *CPU1*. Starting with *to-ilp* on Root, the required attribute values *objective* and *binary-vars* on Root and *to-ilp* on SWRoot and Request are read from cache, since they do not depend on any changed information. Those cached attributes are marked half-grey. Other attributes which normally would be evaluated for those attributes – e.g. *to-ilp* on *Compress* – will never be called. Such skipped attributes are shown in white. They are another reason for the savings of incremental evaluation. However, the attribute *nego* on SWRoot will be re-evaluated, since some attributes it depends on – like *nhw* – need to be re-evaluated. Those attributes are shown in black.

Using this approach, we can incrementally propagate runtime model changes to text or code generated from it, thus affirming **RQ2**. Showing the scalability of our approach, we evaluated models of different sizes and change scenarios.

IV. Evaluation

To answer the research questions **RQ1** and **RQ3** posed in section I, we created synthesized models of systems specified by the grammar described in Section III. More concrete, we strive to measure the benefit of incremental evaluation by comparing our approach to non-incremental versions of it. By using synthesized models it is possible to create arbitrary large systems and, thus, show the suitability and scalability of our approach. The evaluation results and an executable test setup are available at https://github.com/rene-schoene/racr-mquat.

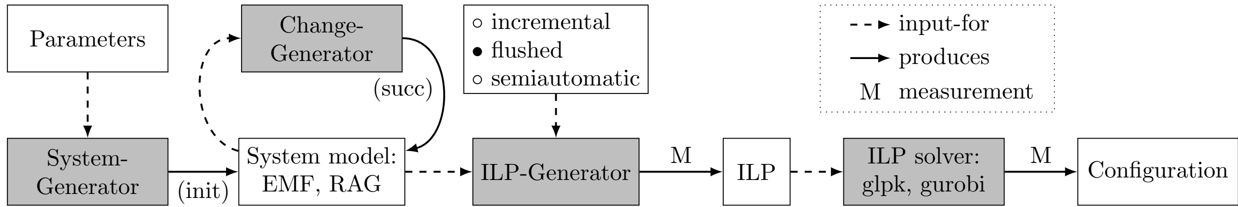


Figure 5: Evaluation setup for measuring ILP generation after initial model creation and successive changes.

A. *Test Setup*

Resources of the created models have two non-functional properties. Software components build a "requirement chain": all but the last component require the next one. In other words, we generate a simplified variant of Pipe-and-Filter architectures [18]. Every clause has its values chosen such that at least one valid solution of the generated optimization problem exists. To enable a close investigation, our test setup provides variability in the following parameters:

(a) number of hardware resources, ranging from 10 to 400,
(b) subresources, either 10 per resource or a flat subtree,
(c) number of software components, ranging from 1 to 90,
(d) implementations per component, 1, 2, or 10, and
(e) modes per implementation, either 2 or 10.

In total, 33 different parameter configurations were used. Based on our experience, we identified three kinds of changes:

Update-HW Update property values in clauses of hardware resources (continuous monitoring)

Structure-HW Remove certain hardware resources, or add new ones (hardware maintenance)

Structure-SW Remove certain software modes, or add new ones (software maintenance)

The category *Update-SW* was left out, because we do not expect a changing behavior of existing implementations. Instead, new software modes or implementations will be added, which is already covered by *Structure-SW*.

To show the applicability of our approach under more realistic conditions, a fourth scenario "mixed" was measured. All three kinds of changes were mixed testing the stability of our approach in contrast to specific, one-sided changes. Based on our experiences, we weighted the kinds with 80% Update-HW, 10% Structure-HW, 5% Structure-SW. In the remaining 5%, there is no change.

Figure 5 explains the characteristics of one measurement. First a model for a given set of parameters is generated. Then, the model is transformed to an ILP, henceforth termed *initial step*. Once the ILP has been generated the first time, a number of changes of a selected kind is applied to the model. After each model change, the ILP transformation is repeated in *successive steps*.

To show the advantage of incremental evaluation, three different strategies for change management are examined, namely *incremental*, *semiautomatic* and *flushed*. Incremental describes our approach and fully utilizes the cache.

Semiautomatic means enabling caching only for auxiliary attributes unrelated to ILP generation and which are most likely manually cached and maintained in a handcrafted solution, e.g., attributes computing a filtered collection containing available hardware resources. In contrast to the actual ILP generation, such auxiliary attributes are simple analyses, whose cache maintenance is simple enough for a handcrafted solution. The semiautomatic strategy is important for realistic measurements, since one usually takes rigorous advantage of such attributes in a sense of calling them many times in algorithm-based analyses. To compare our incremental solution to completely non-cached analyses would be unrealistic and just confirm the well-known fact of exponential attribute evaluation complexity in case of naïve evaluation [19], [20].

The flushed strategy has the cache enabled, but it gets invalidated after each change. Thereby the first invocation of each attribute needs to be computed again, but further invocations in the same step can rely on the attribute being cached. Depending on the context, we will also use the names of each strategy to refer to an execution using this strategy.

For each step, the time required to generate the ILP was measured, excluding disk IO and cache flushing. Additionally, to avoid dependencies of our results to a concrete hardware used to measure, the numbers of called and actually computed attributes were counted. These are indicators, how well the strategy makes use of incremental evaluation, because a lower number of computed attributes implies less computation needed. Using an incremental approach, the set of computed attributes is always a real subset of those evaluated using a non-incremental approach. Using the generation times, we are able to answer **RQ1**, i. e., prove the suitability of our approach. Taking all measured data into account, the benefit of incremental ILP generation is visible, thus answering **RQ3**.

Possible threats to the validity of our evaluation are the following. We only investigated synthesized models due to the need to generate models of many different sizes. Based on our measurements however, we are confident in future applications on realistic MQuat applications of significant size. Further, the number of steps for the first three scenarios is very small, but we believe they are sufficient to show their characteristics. To further counter this point, we added the fourth scenario using all kinds of changes and 100 steps. Finally, we provide no formal correctness proofs in terms of finding an optimal solution. For ILP such proofs are well-known however. A correctness proof therefore boils-down to show that the generated ILP specifications satisfy our intention, in particular in case of incremental generation after model updates.

B. Measured Generation Times

Figure 6a shows the generation time as box plots for eight consecutive hardware value changes and, thus, seven re-generations. Apparently, the incremental strategy performs best for all successive steps, utilizing cached values. Using the flushed, each step needs the same time, as flushing the cache implies a complete re-computation of all attributes. Disabling the cache for all generation-related attributes leads to a similar picture, however for different reasons: The initial step takes longer on average than successive steps, as auxiliary attributes are still cached. Those cache entries need to be created, but can be used on successive steps to decrease execution time. The incremental strategy clearly takes less time for generation, as it is on average faster by a factor of 4.81 and 5.69 compared to semiautomatic and flushed, respectively.

Figure 6b depicts the generation times when changing the hardware structure. Each even step removes some resources from the model, whereas each odd, successive step adds the removed resources again. Flushed runs longer than semiautomatic for the second and third resource removing step, because of cache updates of auxiliary attributes. For all other steps, semiautomatic lags behind flushed for the same reasons as in the Update-HW case. With the incremental strategy all successive steps can be completed faster compared to the other strategies, because of the unrestrained use of cached values.

Figure 6c unveils the shortcoming of current attribution, even when using the incremental strategy. In the second to fourth step, new modes are added, which are removed together with existing modes in the next three steps. For all strategies, an increase of the generation time is visible when modes are added. However, the order of measured generation times always stays the same: flushed is faster than semiautomatic and slower than incremental. The higher computation times of structural changes compared to Update-HW are the result of newly computed constraints for hardware resources and new parts of constraints for new software modes.

In Figure 7, the generation times for the *mixed* scenario along with their averages are depicted for all three different strategies. The observed execution time is shown using box plots, as all different combinations of parameters are shown in this figure. For semiautomatic and incremental, only the initial step takes longer, because a cache needs to be built up (auxiliary attributes for semiautomatic). Although all attributes are actually cached in the incremental strategy, this initial step needs less time compared to semiautomatic, because cache-hits of ILP-related attributes occur in this initial step and reduce computation. Those observations cannot be made for flushed. Generation times always stay about equal except after a change adding new software modes leading to more computation and increases all following generation times. They stay equal, because the cache is flushed right after the change, leading to a similar computation effort in each step.

Note, that the initial step of flushed and incremental takes the same amount of time, as exactly the same computation is done. The cache is flushed after generating the ILP, thereby not influencing the first measured time. The most obvious difference of Figure 7c compared to the other two strategies is the small generation time for all non-structural changes (updating resource properties and no change). For structural changes, more computation time is needed, as stated above. All effects lead to average times of 3.41, 22.43, and 27.79 seconds for incremental, flushed and semiautomatic, respectively.

C. Results for Attribute Metrics

In addition to generation times, we evaluated the following attribute metrics. Below, $comp_N$ and $called_N$ denote the number of computed and called attributes for strategy N, respectively. Notably, $N \in \{1, 2\}$ denotes two strategies, which are compared with each other.

Cache Miss Rate $= \frac{comp_1}{called_1}$. This shows the degree of "incrementality" for each strategy, i.e., how good the cache entries are reused on average. Lower values are better.

Cache Potential $= \frac{comp_1}{called_2}$. The second metric shows the potential increase of cache usage when using one strategy over another and can be compared to the cache miss rate.

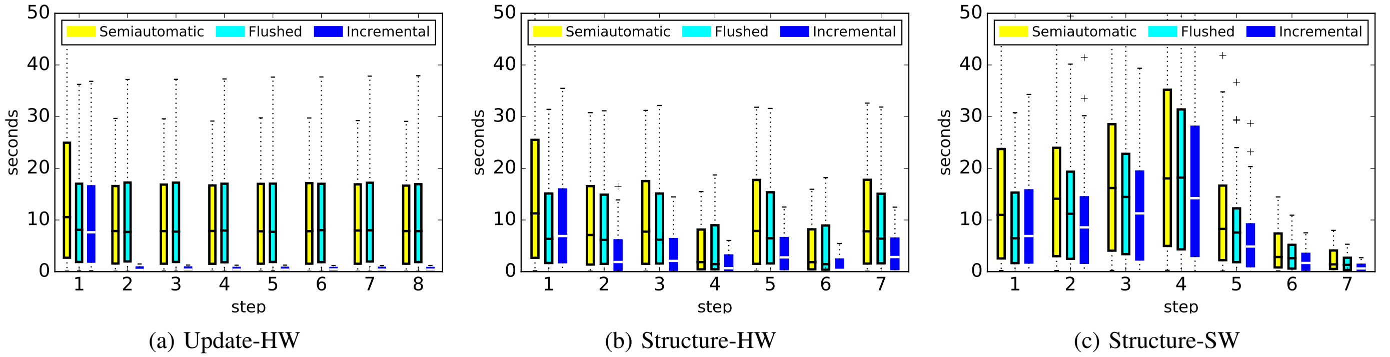
Speed-Up $= \frac{comp_2}{called_2} - \frac{comp_1}{called_2} = \frac{comp_2 - comp_1}{called_2}$. This is the difference of the second strategy's cache miss rate and potential to the first. It shows the gain of using one strategy over another. A higher value indicates that fewer attributes need to be computed using the first strategy.

Table I shows the average for each attribute metric, scenario and strategy (<u>inc</u>remental, flushed, <u>semi</u>automatic). Concerning cache miss rates, the incremental approach is obviously always ahead of the other strategies. This is because the cache is maintained between changes (in contrast to flushed) and used for each attribute (in contrast to semiautomatic). Flushing the cache between changes leads to a drop of about 7.8% on average. Using no cache apparently results in a cache miss rate of 1.0, as every called attribute is also computed.

When using the incremental approach, fewer attributes get called in the first place. This is because calls to dependent attributes can be skipped, if the value of the calling attribute is read from cache. As an example, in the biggest mixed-scenario attributes were called 10 135 309 times in total while using incremental, 41 106 310 with flushed and 82 660 090 using the semiautomatic strategy.

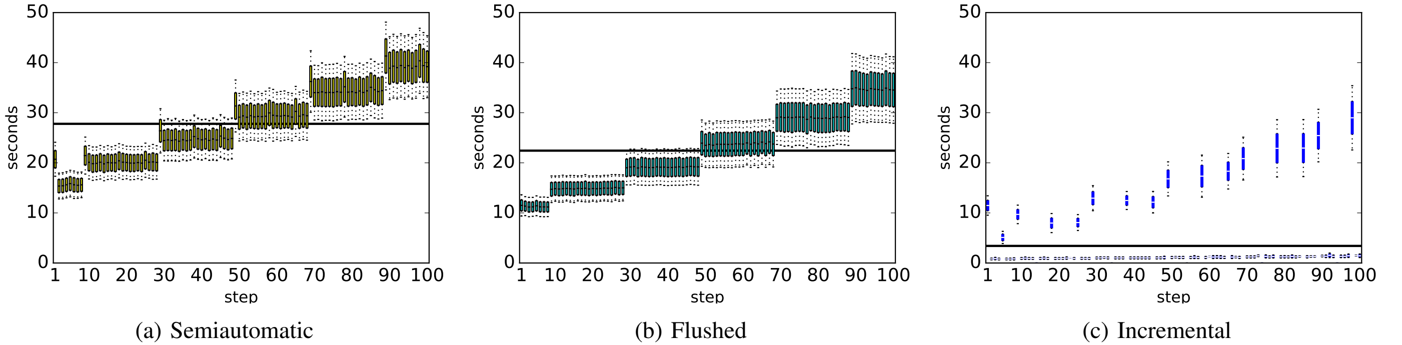
Regarding cache potential of flushed to incremental, values range from 2.8% for mixed to 32.3% for Structure-SW. This indicates, that in the latter case, our approach has less impact. Values of semiautomatic to incremental are even lower, indicating more computations can be saved using our approach.

For speed-ups, one can observe two effects. The first includes speed-ups from semiautomatic, which in every case is on average between 82 and 90%. This can be read as a direct reduction of computation, i.e., using either incremental or flushed saves those amounts of attribute computations. Hence, this also results in lower execution times. In the "mixed" scenario, which includes all kinds of changes, our technique still provides good results with average savings of 16%.



(a) Update-HW (b) Structure-HW (c) Structure-SW

Figure 6: Generation times for each step, i.e., after a change of the model, using different strategies and kinds of changes



(a) Semiautomatic (b) Flushed (c) Incremental

Figure 7: Generation times for each step, i.e., after a change of the model, and averages (bold line) for the *mixed* scenario

Strategy	inc	flushed	semi	inc→flushed		inc→semi		flushed→semi	
Metric	Cache Miss Rate (%)			Pot	Sp	Pot	Sp	Pot	Sp
Update-HW	0.210	0.319	1.0	0.056	0.263	0.031	0.969	0.177	0.823
Structure-HW	0.195	0.321	1.0	0.154	0.166	0.086	0.914	0.179	0.821
Structure-SW	0.330	0.341	1.0	0.323	0.018	0.171	0.829	0.181	0.819
Mixed	0.116	0.335	1.0	0.029	0.307	0.014	0.986	0.167	0.833

Table I: Attribute metrics showing Cache Miss Rate, *Pot*ential and *Sp*eed-up

D. Conclusion of the Evaluation

We have shown the scalability of our approach, saving 35 to 87% w.r.t. processing time and 1.8 to 16% w.r.t. cache usage. With that, we have shown the benefit asked by our initial research question **RQ3**.

Besides the shown advantages of our presented solution, the usage of RAGs has further advantages, which are difficult to quantize. First, the model transformation is specified declaratively. This enhances maintainability and modularity, as attributes can be seen as aspects that can be changed independently [21]. Second, RAGs and term rewriting have a formal basis [22], [11], which can be used to verify properties like termination or complexity. Finally, like in our previous work [23], cache-consistency and efficiency is automatically ensured by our approach, as attributes are recomputed only if the values they dependent on changed.

A disadvantage using synthesized models is the loose link to real examples, which is to some degree mitigated by the diversity of created models and applied changes. However, it was mandatory to prove scalability.

V. Related Work

This work combines several research areas, including self-adaptive systems and models at runtime, incremental computation, and model transformation.

a) Self-adaptive Systems: There are plenty of approaches to design self-adaptive systems, providing frameworks and middleware. Most of these systems, such as MUSIC [24], DiVA [6] or ConFract [25], use models to describe their state and context. For a better distinction, we classify our solution using a survey in the field of self-adaptive systems [26], which aims at building a taxonomy for engineering such systems. Our approach adapts reactively, i.e., it reacts to previous events. The reason for adaptation is the user, managed elements are software components and the controlling system is a middleware. While the kind of changes is currently compositional, one could also think of parameter adaptation. As already mentioned in Section II, the adaptation logic is external, criteria for decision are models, and our approach

is centralized, because *RACR* is not yet developed to be a distributed application.

In [9] and [27], the developers of KMF aim at gathering requirements for runtime models and comparing their work to EMF. They consider a small memory footprint, efficient model navigation and thread-safety. In [9], caching was proposed as a promising solution which can cause problems such as inconsistency. Using RAG-controlled rewriting, caching is provided with the guarantee of being consistent.

There are a few works on reasoning at runtime having a specific focus on efficiency. One work is about time-distorted reasoning at runtime [28], where KMF is used to efficiently handle changes in runtime models. They do not follow the standard approach of having multiple snapshots of the model for different times, but instead all information is stored in one model. Every model element has special operations to shift between versions of it in time. The main difference to our approach is, that they analyze using historical data and a standard imperative language. In this work we only known about the current state, but can use already computed intermediate results from unchanged parts of the model.

b) Incremental Evaluation: Incremental evaluation is the "efficient recomputation in response to changes in input data" [29]. It is also called "self-adjustable computing" in the context of imperative programs tracking data and control dependencies [30]. Most works in this field agree on the advantages, such as efficiency, gained at cost of higher memory consumption [31], [32]. However, they propose either language extensions or new languages with the need to explicitly wrap access to data. Using RAG-controlled rewriting, the way of accessing data is not changed, but caching is provided automatically.

c) Model Transformation: Model Transformation is an active field of research with many approaches mostly targeting design time. For a better comparison of this work, we classify our approach using a taxonomy [33]: Source and target model in our approach are both hierarchical, from different technical spaces and exogenous. The transformation is one-to-one, horizontal, syntactical, fully automatic, complex and information-preserving. Using *RACR*, the transformation is CRUD-changeable, without suggestions and highly reusable. The transformation is tested and has neither inconsistency handling, generality of rules nor bi-directionality. However, rules are decomposable and support change propagation. It is designed to be scalable and extensible at design time, but not yet interoperable nor widely used within the community.

Another survey covering model-to-text (M2T) transformations [34] comprises three categories for comparison: language coverage, execution phases and overhead. *Racr-mquat* supports arbitrary input and output elements, NACs as well as PACs, but no imperative parts. Source minimality, detection at every granularity, atomic inserts and deletes are ensured by *RACR*. As our approach exploits lazy evaluation, change log optimization is supported indirectly. Traces are automatically computed for Model2Transformation at runtime. Change propagation is partial, unidirectional, at the level of rules and sequentially. Overheads are mostly for memory, minor runtime and not specification.

Another approach is eMOFLON [35], which is also included in the previous survey and based on Triple Graph Grammars and the Eclipse Modeling Framework. It supports bidirectional transformation, but without support for incremental evaluation.

Another approach of Kusel [34] is EMF-IncQuery [36], which combines the query language VIATRA2 with EMF models. It caches query results and updates those values leveraging the notification mechanism provided by EMF. In [37], template-based M2T transformations are extended by adding *signatures* to it, such that templates are only computed if their corresponding signature has changed. However, signatures can either be generated automatically, sometimes not covering the complete dependencies, or manually, which is time-consuming and error-prone [37]. Using our approach, such dependencies are automatically calculated and ensured at runtime.

Bergmann et. al showed incremental model transformation [38] using a RETE-based pattern matcher storing matched left hand sides and update them incrementally upon changes. The main difference is, that our approach is only applicable to trees with overlay graphs, but provides much better support for complex analyses and not just plain transformations.

VI. Conclusion and Future Work

In this work, we enumerated several challenges concerning the use of runtime models with a particular focus on scalability. After a brief introduction of previous work, we described our approach involving the use of references attribute grammars to model and reason about the state of a system. We showed, that our approach, RAG-controlled rewriting, is suitable to describe hierarchical runtime models and corresponding analyses using attributes. Furthermore, those analyses are incrementally evaluated, as only those parts of the analysis affected by model updates are re-evaluated. Hence, unnecessary and possibly costly evaluations can be skipped.

Using synthesized models of varying size, we achieved a speed-up of 35 to 87% w.r.t. processing time and 1.8 to 16% w.r.t. cache usage. Thus, it is possible to efficiently use runtime models within self-adaptive systems despite many small changes affecting only parts of those models. In addition, runtime models, their analyses and updates are declaratively specified using well-founded metacompiler technologies.

In the near future, we plan to investigate models of bigger and realistic use cases, evaluate the memory usage and use our approach to develop practical MQuAT applications. Furthermore, we strive to implement existing benchmark cases, like the TTC 2015 Train Benchmark Case [39] to allow for better comparison with other tools. In the long term, we envision a heuristic approach for solving the optimization problem described in Section 2 completely implemented using *RACR*, thus fully utilizing incremental evaluation.

Acknowledgments

This work is partly supported by the German Research Foundation (DFG) within the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing" and the cluster of excellence cfaed.

References

[1] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *Computer*, vol. 42, no. 10, Oct. 2009.

[2] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, Jan. 2003.

[3] M. Broy, M. V. Cengarle, and E. Geisberger, "Cyber-physical systems: Imminent challenges," in *Large-Scale Complex IT Systems: Development, Operation and Management*, ser. LNCS, vol. 7539. Springer, 2012.

[4] R. Lemos *et al.*, "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS, vol. 7475. Springer, 2013.

[5] S. Götz, "Multi-Quality Auto-Tuning by Contract Negotiation," PhD Thesis, Technische Universität Dresden, Apr. 2013.

[6] F. Fleurey and A. Solberg, "A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems," in *Model Driven Engineering Languages and Systems*, ser. LNCS, vol. 5795. Springer, Oct. 2009.

[7] Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar, "Implicit Self-adjusting Computation for Purely Functional Programs," in *ICFP*. ACM, 2011.

[8] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé, *Vision and challenges for realising the Internet of Things*. EUR-OP, 2010.

[9] F. Francois *et al.*, "Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use," University of Luxembourg, Tech. Rep. TR-SnT-2014-11, May 2014.

[10] C. Bürger, "Reference Attribute Grammar Controlled Graph Rewriting: Motivation and Overview," in *Software Language Engineering: 8th International Conference*. ACM, 2015.

[11] G. Hedin, "Reference attributed grammars," *Informatica (Slovenia)*, vol. 24, no. 3, 2000.

[12] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, 1968.

[13] U. Aßmann *et al.*, "A Reference Architecture and Roadmap for Models@run.time Systems," in *Models@run.time: Foundations, Applications, and Roadmaps*, ser. LNCS, vol. 8378. Springer, 2014.

[14] L. A. Wolsey, *Integer programming*. John Wiley & Sons, 1998, vol. 42.

[15] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, 1990.

[16] T. Reps, "Generating language-based environments," Cornell University, Tech. Rep., 1982.

[17] C. Bürger, S. Karol, C. Wende, and U. Aßmann, "Reference attribute grammars for metamodel semantics," in *Software Language Engineering*, ser. LNCS, vol. 6563. Springer, 2011.

[18] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice Hall Englewood Cliffs, 1996, vol. 1.

[19] M. Jourdan, "An optimal-time recursive evaluator for attribute grammars," in *International Symposium on Programming: 6th Colloquium*, ser. LNCS, vol. 167. Springer, Apr. 1984.

[20] J. Paakki, "Attribute grammar paradigms—a high-level methodology in language implementation," *ACM Computing Surveys (CSUR)*, vol. 27, no. 2, 1995.

[21] P. Avgustinov, T. Ekman, and J. Tibble, "Modularity first: A case for mixing aop and attribute grammars," in *AOSD '08: Proceedings of the 7th International Conference on Aspect-Oriented Software Development*. ACM, Apr. 2008.

[22] F. Baader and T. Nipkow, *Term rewriting and all that*. Cambridge university press, 1999.

[23] C. Bürger *et al.*, "Using Reference Attribute Grammar-Controlled Rewriting for Energy Auto-Tuning," in *10th International Workshop on Models@run.time*, Sep. 2015.

[24] M. Alia *et al.*, "A Component-Based Planning Framework for Adaptive Systems," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, ser. LNCS, vol. 4276. Springer, Oct. 2006.

[25] H. Chang, P. Collet, A. Ozanne, and N. Rivierre, "From Components to Autonomic Elements Using Negotiable Contracts," in *Autonomic and Trusted Computing*, ser. LNCS, vol. 4158. Springer, Sep. 2006.

[26] C. Krupitzer *et al.*, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, Part B, Feb. 2015.

[27] F. Fouquet *et al.*, *An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements*, ser. LNCS. Springer, 2012, vol. 7590.

[28] T. Hartmann *et al.*, "Reasoning at Runtime using time-distorted Contexts: A Models@run.time based Approach," in *26th International Conference on Software Engineering and Knowledge Engineering (SEKE'14)*. Knowledge Systems Institute Graduate School, USA, Jul. 2014.

[29] P. J. Guo and D. Engler, "Towards Practical Incremental Recomputation for Scientists: An Implementation for the Python Language," in *Proceedings of the 2nd Conference on Theory and Practice of Provenance*, ser. TAPP'10. USENIX Association, 2010.

[30] S. Burckhardt *et al.*, "Two for the Price of One: A Model for Parallel and Incremental Computation," in *OOPSLA '11*. ACM, 2011.

[31] S. E. Hudson, "Incremental attribute evaluation: A flexible algorithm for lazy update," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 3, 1991.

[32] U. A. Acar, "Self-Adjusting Computation," PhD Thesis, Carnegie Mellon University, May 2005.

[33] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, Mar. 2006.

[34] A. Kusel *et al.*, "A Survey on Incremental Model Transformation Approaches," in *Models and Evolution Workshop*. CEUR-WS.org, 2013.

[35] A. Anjorin, M. Lauder, S. Patzina, and A. Schürr, "eMoflon: Leveraging EMF and Professional CASE Tools," *Informatik*, 2011.

[36] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, "A graph query language for EMF models," in *Theory and Practice of Model Transformations*, ser. LNCS, vol. 6707. Springer, 2011.

[37] B. Ogunyomi, L. M. Rose, and D. S. Kolovos, "On the Use of Signatures for Source Incremental Model-to-text Transformation," in *Model-Driven Engineering Languages and Systems*, ser. LNCS, vol. 8767. Springer, 2014.

[38] G. Bergmann *et al.*, "Incremental Pattern Matching in the Viatra Model Transformation System," in *Proceedings of the Third International Workshop on Graph and Model Transformations*. ACM, 2008.

[39] G. Szárnyas, O. Semeráth, I. Ráth, and D. Varró, "The TTC 2015 Train Benchmark Case for Incremental Model Validation," *8th Transformation Tool Contest (TTC 2015)*, 2015.