Using Reference Attribute Grammar-Controlled Rewriting for Energy Auto-Tuning

Christoff Bürger[1], Johannes Mey[2], René Schöne[2], Sven Karol[3], and Daniel Langner[2]

[1] Department of Computer Science, Faculty of Engineering, Lund University, Sweden
`christoff.burger@cs.lth.se`

[2] Professur Softwaretechnologie, Institut für Software- und Multimediatechnik, Fakultät Informatik, Technische Universität Dresden, Germany
`(johannes.mey@|rene.schoene@|daniel.langner@mailbox.)tu-dresden.de`

[3] Chair for Compiler Construction, Center for Advancing Electronics Dresden, Technische Universität Dresden, Germany
`sven.karol@tu-dresden.de`

Abstract. Cyber-physical systems react on events reported by sensors and interact with objects of the real world according to their current state and their view of the world. This view is naturally represented by a model which is continuously analysed and updated at runtime. Model analyses should be ideally concise *and* efficient, requiring well-founded, comprehensible implementations with efficient reasoning mechanisms. In this paper, we apply *reference attribute grammar controlled rewriting* to concisely implement the runtime model of an auto-tuning case study for energy optimization. Attribute functions are used to interactively perform analyses. In case of an update, our system incrementally—and, thus, efficiently—recomputes depending analyses. Since reference attribute grammar controlled rewriting builds the required dependency graphs automatically, incremental analysis comes for free.

Keywords: energy auto-tuning, cyber-physical system, runtime model, attribute grammar, graph rewriting, incremental analyses

1 Introduction

Developing software for cyber-physical systems is a major challenge in software and systems engineering [2,4,11]. A cyber-physical system steadily monitors its environment, interacts with real-world entities and other systems, and reconfigures itself according to its current view of the world. To implement these tasks, executable *world models* enriched with runtime analyses and update events (e.g, from sensors) are a common approach [3,19]. To ensure reliability and integrity of cyber-physical systems, such models need to be well-founded, interactive and comprehensible. In particular, model analyses (e.g., to react on events accordingly) should not only be easy to implement but also *efficient*—especially in the presence of continuous model updates at runtime [1]. Hence, an important concern is to

avoid unnecessary or redundant reevaluations of the analyses. Essentially, if not affected by updates, previously computed results should be reused in such a way that model analyses become incremental. A second challenge for cyber-physical systems is the energy scarcity, demanding for energy-efficient systems [2].

In this paper, we demonstrate how a novel combination of reference attribute grammars [10] and rewriting [14,17]—both well-known compiler-compiler techniques [17,9,13]—can improve on the challenge of incremental model analysis. *Reference attribute grammar controlled rewriting* (RAG-controlled rewriting) [6] leverages dynamic dependency tracking to make attribute evaluation (i.e., model analyses) incremental in the presence of rewrite-based model updates. We present an energy auto-tuning case study whose runtime model is incrementally updated using RAG-controlled rewriting. Our example scenario is an interactively triggered energy-efficient indexing of text documents that runs on a changeable network of independently powered computers.

The paper is structured as follows: first, the energy auto-tuning scenario of our case study is presented (Section 2), followed by a short introduction to RAG-controlled rewriting (Section 3); afterwards, the implementation of the actual runtime model based on RAG-controlled rewriting is presented (Section 4); an evaluation of the presented solution concludes the paper (Section 5).

2 Case Study: Energy-efficient Document Indexing

Case Study Overview: Our case study to evaluate the applicability of RAG-controlled rewriting for runtime models is the energy-efficient indexing of text documents using a network of embedded computers. Indexing a text document in our context means to construct a hash-table that counts the frequency of each word in the document. A typical application of this indexer may be instant search or case-based reasoning in a network of robots [20]. Each indexing request has two parameters: a text document of a certain size to index and a hard deadline until which the result must be computed. Indexing requests are interactive (i.e., occur randomly); at any moment, only previous requests are known but nothing about future requests, including their frequency, deadlines and document sizes. Given an arbitrary stream of indexing requests and a network of independent computers available for indexing (called workers), the objective is to dispatch the processing of requests in such a way that the total energy consumed to process the stream is minimized.

To save energy, single devices or complete network parts—including their switches, attached workers and subnetworks—can be deactivated. Furthermore, indexing can be prolonged as long as the corresponding deadlines are met. All devices are exclusively available for indexing and not occupied with other tasks. The start-up and shut-down times of devices are not constant however; sometimes toggling them may even fail and require retries[4]. Furthermore, devices can be removed interactively or new ones added, requiring reschedules to still meet deadlines and optimise energy consumption.

[4] This is a realistic problem that happens with some of our devices (*CubieBoards*).

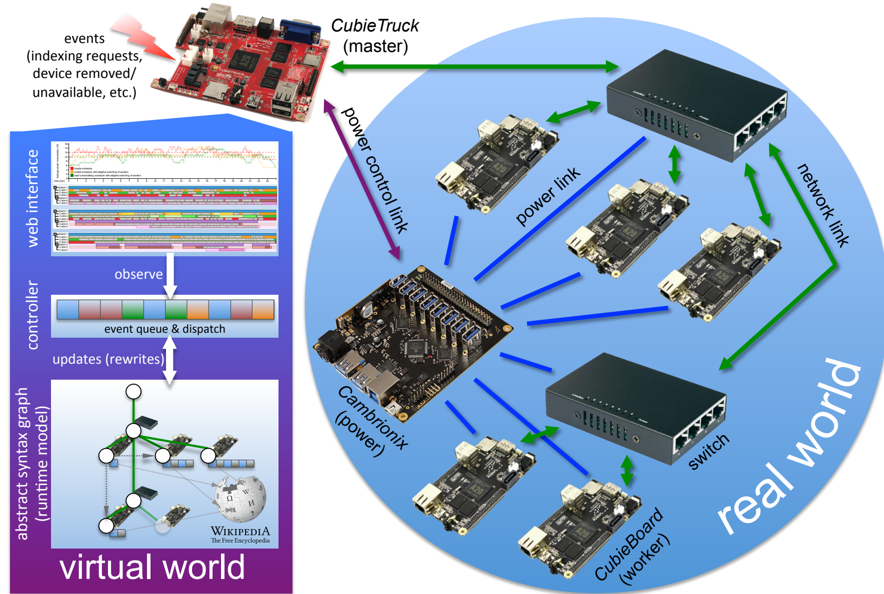


Fig. 1: Cyber-physical system of the indexing case study.

Setup and Solution Overview: Figure 1 summarises the hard- and software components of our solution. The hardware (real-world circle) comprises two switches (*Conrad FX-08Mini*: 8 RJ45 ports à 100 MBit/s), five worker devices (*CubieBoard2*: *Allwinner A20* (*ARM Cortex-A7 Dual Core*), 1GB DDR3), a remote-controllable power supply (*Cambrionix U8S*: 8 USB ports a max. 5.2V, 2.1A) and a master device (*CubieTruck*: *Allwinner A20*, 2GB DDR3). Each port of the power supply powers a single switch or worker (blue power links), and is controlled by the master for power measurement and toggling (purple power control link). The two switches are connected in sequence (green network links): the first is directly connected to the master and three workers; the second to two.

The software includes a controller that dispatches events and interacts with devices (control loop), a runtime model including analyses to schedule indexing requests and deduce reconfigurations, a web interface to visualise power consumption and schedules, a concurrent implementation of an actual indexing algorithm, and communication links between master and both, workers and power supply.

The controller, runtime model and web interface run on the master (Fig. 1, left side). Thus, only the master has a view on the real world and can interact with it. It is responsible for runtime model updates and reconfigurations, i.e. the propagation of such an update into the real world. All other devices have no knowledge about the world's current state. Moreover, the actual indexing requests issued by external entities are exclusive events to the master. Because these requests occur concurrently, they are collected in a queue by the controller for dispatching. The web interface observes this queue and visualises its history.

Three kinds of instruction events and respective acknowledgements for device communication exist: to start indexing of a text document on a certain worker (master ↔ worker), to start-up/shut-down devices (master ↔ power supply and/or workers) and power measurements (master ↔ power supply). Devices are either idle and ready for instructions, or busy, in which case they cannot process new instructions (indexing cannot be suspended and each worker processes at most one request at a time, thus resulting in a virtual request queue stored in the runtime model). If ready, received instructions are immediately performed; their start and result is acknowledged to the master.

The controller performs two main tasks in its control loop. First, it dispatches the event queue and updates the runtime model according to acknowledgement events; in case of indexing requests it delegates to the runtime model to incorporate a schedule (such updates are provided by the runtime model in the form of rewrite rules, cf. Section 4). Second, it periodically checks for devices that can be shut down, indexing tasks that must be started, and new or failed devices. These periodic tasks use runtime model analyses for reasoning.

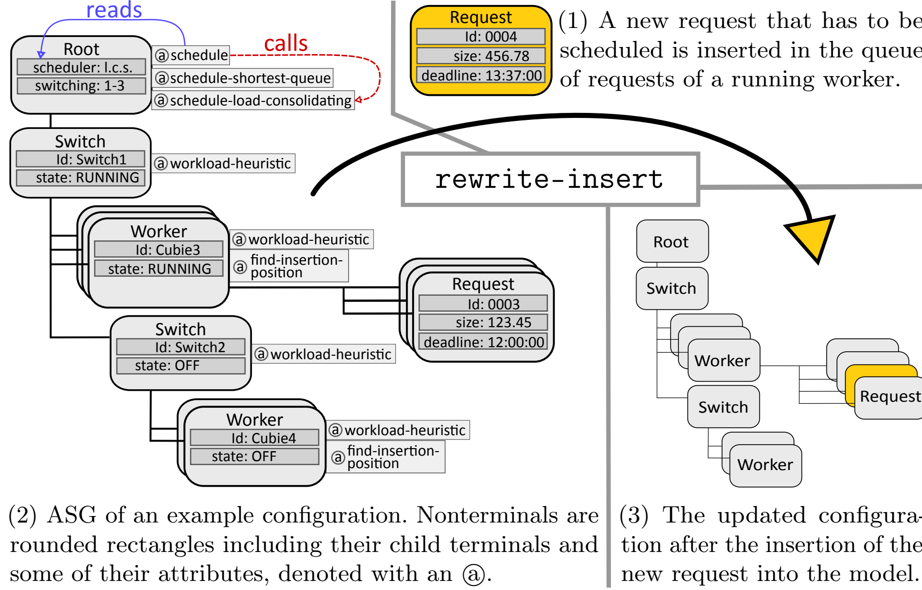
Our runtime model combines a scheduling and energy policy. Consolidating workload, the scheduling satisfies deadlines using as less resources as possible. The energy analysis deduces unused, and in the future most likely not required, devices—it saves energy by shutting them down taking into account reactivity for emerging requests (energy-awareness). Both analyses are well-balanced; it is their combination that saves energy while still processing as many indexing request as possible (cf. Section 5).

3 RAG-Controlled Rewriting Background

A well-known formalism for the specification of semantic analyses are attribute grammars [15]. Their main advantage is to relieve users from manual traversals of analysed models and scheduling of analyses. Given a concrete attribute grammar, an highly optimised evaluator can be generated based on a static analysis of the grammar's attribute dependencies [16]; essentially, incremental model analysis is achieved. However, attribute grammars are specified w.r.t. tree structure, for which reason their graph analysis capabilities are inconvenient.

Reference attribute grammars (RAGs) improve on this by extending attribute grammars with so called reference attributes, attributes evaluating to nodes of the tree being analysed [10], enabling the deduction of abstract syntax graphs (ASGs) from abstract syntax trees (ASTs) [7] and reasoning about ASGs [12]. The viability of this technique to specify semantics for metamodels, like *EMF Ecore* [8], has been shown [7]. The main disadvantage of RAGs is, that reference attributes hinder static dependency analyses and therefore incremental evaluation.

To still achieve incremental evaluation, recent research proposed the tracking of attribute dependencies at evaluation time, i.e., dynamically [6,18]. We even go one step further and comprehend AST changes in terms of graph rewriting [6], thus combining incremental RAG-based analyses and rewriting-based model transformations, either interactive, deduced or a combination thereof.



(1) A new request that has to be scheduled is inserted in the queue of requests of a running worker.

(2) ASG of an example configuration. Nonterminals are rounded rectangles including their child terminals and some of their attributes, denoted with an @.

(3) The updated configuration after the insertion of the new request into the model.

Fig. 2: Upon arrival of a new request (1), the attribute *schedule* is evaluated (2) and the request is inserted into the queue of a worker (3).

4 A Runtime Model based on RAGs and Rewriting

In the following, we present a RAG-controlled rewriting-based runtime model for the case study of Section 2. Our implementation uses *RACR* [5,6], a reference implementation of RAG-controlled rewriting in *Scheme*[5]. Using RAGs, we represent the runtime model (cf. Fig. 1) as an abstract syntax graph (ASG) with an underlying abstract syntax tree (AST) as a distinct spanning tree. Using *RACR*, we define the corresponding AST grammar as follows:

```
1 (ast-rule 'Root->scheduler-backupworkers-CompositeWorker)
2 (ast-rule 'AbstractWorker->id-state-timestamp)
3 (ast-rule 'CompositeWorker:AbstractWorker->AbstractWorker*)
4 (ast-rule 'Switch:CompositeWorker->)
5 (ast-rule 'Worker:AbstractWorker->devicetype-Request*<Queue)
6 (ast-rule 'Request->id-size-deadline-dispatchtime)
```

Each line describes one nonterminal that corresponds to a possible node type in the AST. For example, Line 5 defines a *Worker* inheriting from *AbstractWorker* and therefore comprising an ID, a current state, the most recent alternation date, the type of worker and any number of *Requests* called *Queue*. A concrete ASG (i.e., an AST with evaluated attributes) is shown in Fig. 2b.

Analyses are concisely specified using attributes. The main attribute *schedule* of the scheduling algorithm is depicted below while its evaluation is exemplified in

[5] The implementation of our solution can be found at `https://github.com/2bt/haec`.

the upper part of Fig. 2b. The attribute is defined for *Root*—the start symbol in the AST grammar, which defines virtual root components in our runtime model, and does a dynamic dispatch to determine which scheduler to use. It reads the terminal *scheduler* using *ast-child* and invokes the another attribute via *att-value*.

```
1 (ag-rule schedule
2   (Root (lambda (n time work-id load-size deadline)
3     (att-value (ast-child 'scheduler n) n time work-id
4                 load-size deadline))))
```

We employ two basic *scheduling strategies*. The first one always chooses the worker with the shortest queue (*shortest-queue scheduler*) whereas the second tries to fill the request queues of running workers as much as possible in order to consolidate the load (*load-consolidating scheduler*). Another analysis is the *energy-aware adaptation strategy*, which periodically computes the ideal number of online workers and switches off everything else, trying to keep at most one worker with an empty queue running as backup for further incoming tasks.

Figure 2 shows an example of a rewrite that creates a new *Request* and inserts it in the request queue of a selected worker. In *RACR*, such model transformations can be specified as below. After each rewrite application, the cache of every attribute depending on values of the rewritten part of the ASG is automatically flushed, flagging those attributes for reevaluation. The actual computation of the new attribute value is *lazy*, which prevents unnecessary work and thus enables fast, incremental analyses of the runtime model.

```
1 (rewrite-insert
2   (ast-child 'Queue worker) ;list-node to insert into
3   index ;position of insertion
4   (create-ast spec 'Request (list id size deadline #f)))
```

An additional use case for rewrites is the failure of a physical worker. In this case, the current and all pending requests are removed from the worker in the ASG and simply scheduled again—now without considering the failed worker.

As described in Section 2, we use a socket-based communication protocol between the devices. Figure 3 sketches the processing of a request. Upon arrival at the controller, a request event is created and added to the runtime model, where a scheduler is used to determine a processing worker. A command to process the request is first sent to the controller, which relays it to the appropriate worker. Once finished, the controller is notified and invokes *event-work-complete* to let the runtime model update accordingly.

5 Evaluation

To evaluate the presented approach, the applied measurement techniques are introduced, followed by a discussion of the acquired data. These very use-case specific results are followed by a more general investigation of the applicability of RAG-controlled rewriting for runtime models.

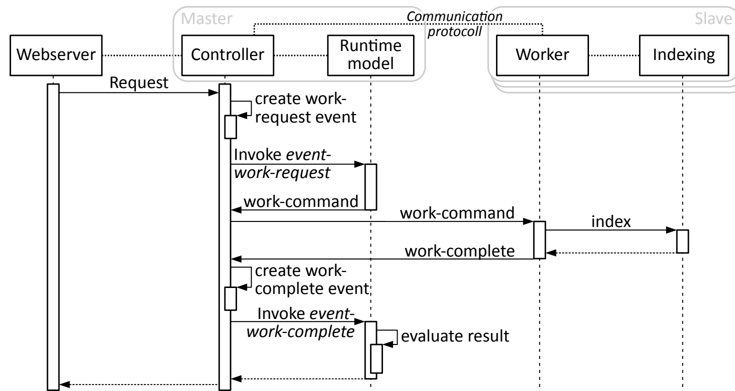
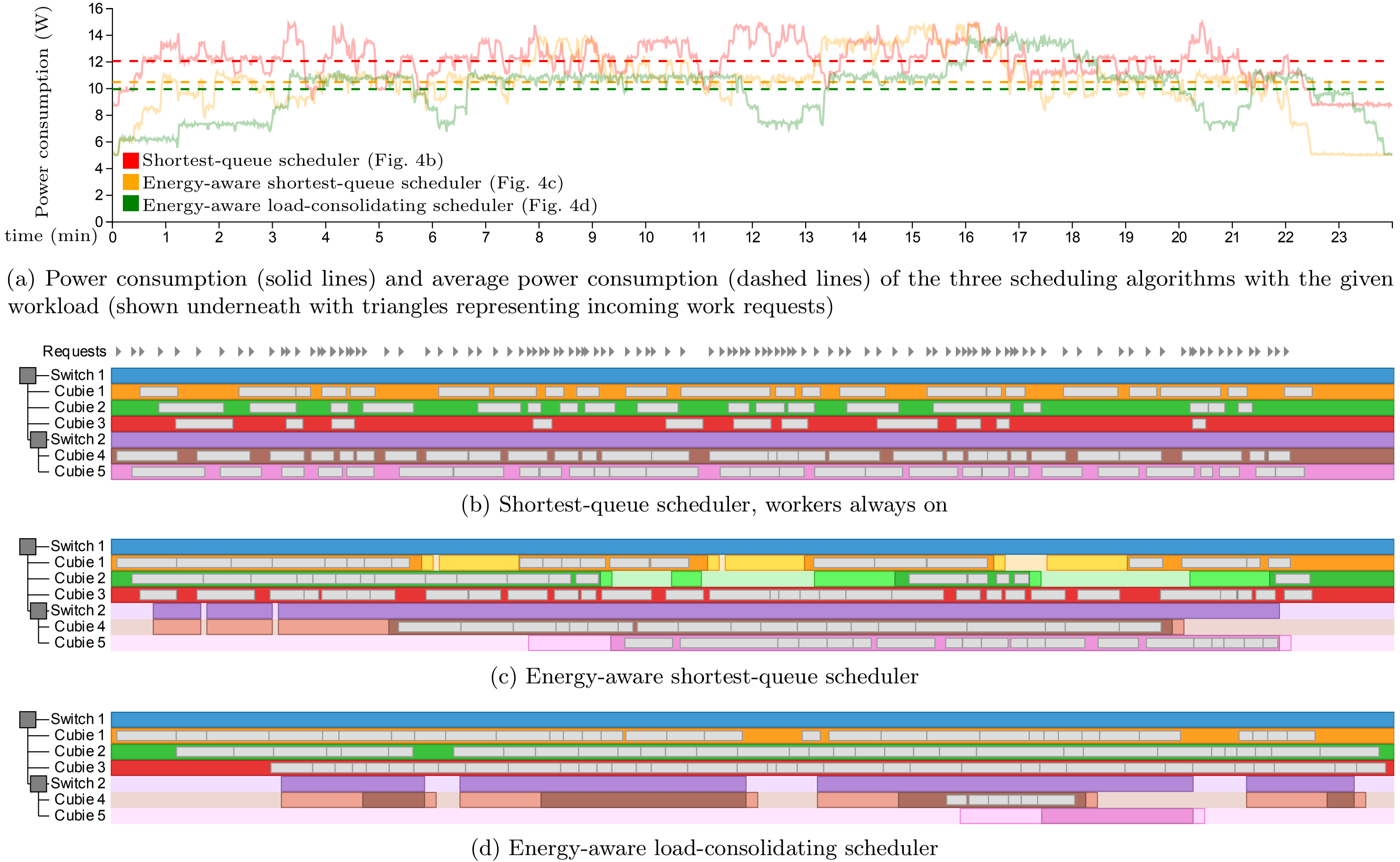


Fig. 3: After arrival of a request, the controller dispatches events.

Performance and Energy Savings As mentioned in Section 2, the controller constantly monitors the state of the workers and their energy consumption, and protocols them as well as the events it processes. For the evaluation, periods of running the same workload in different system configurations have been selected to compare the energy-consumption properties of them. Figure 4 shows three different runs using both implemented scheduling and worker adaptation strategies. The bar charts associated with the three runs show how the same set of work requests (displayed as grey triangles) are scheduled. While the shortest-queue schedulers in Fig. 4b and 4c schedule events to run as soon as possible, the load-consolidating scheduler tries to fill the request queue as much as possible, resulting in less workers being used. Additionally, runs in Fig. 4c and 4d use the energy-aware adaptation strategy (cf. Section 4) with three workers kept running to handle the assumed base load. Each device has a lane in the diagram, the brightness of its colour determines its state: very bright means the worker is switched off, fairly bright means it is currently booting or shutting down, and a dark colour represents an active worker connected to the master. Currently processed work requests are shown as grey boxes on the respective worker.

The power consumption of all three runs is shown in Fig. 4a. A dashed line depicts the average power consumption which is proportional to the required energy. As summarised in Table 1, with the best heuristics used in run 4d, only 82.5% of the simple scheduling heuristics are used, amounting to savings of 17.5%. An important effect of the load-consolidating scheduler is that less workers are needed while the deadlines are kept.

Benefits of a RAG-based Runtime Model Besides the energy savings described in the last section, a RAG-controlled rewriting-based approach offers qualitative benefits for developers. Even though context-free grammars describe tree structures, RAGs can express arbitrary graphs through their reference attributes. No name resolution tables have to be defined explicitly and navigation in a RAG-based model can be performed easily. Furthermore, the declarative



(a) Power consumption (solid lines) and average power consumption (dashed lines) of the three scheduling algorithms with the given workload (shown underneath with triangles representing incoming work requests)

(b) Shortest-queue scheduler, workers always on

(c) Energy-aware shortest-queue scheduler

(d) Energy-aware load-consolidating scheduler

Fig. 4: Comparison of the power consumption using different scheduling algorithms with the same workload

Scheduler	Energy (J)	Avg. Power (W)	Rel. Energy (%)
Shortest-queue	17,378.2	12.07	100.0
Energy-aware, shortest-queue	15,106.9	10.49	86.9
Energy-aware, load-consolidating	14,343.9	9.96	82.5

Table 1: Energy savings overview.

Component	files	blank	comment	code
Webserver	8	484	104	1823
Controller	14	342	28	1217
Client Controller	3	74	17	231
Indexing Task	1	67	11	263
Runtime Model	1	71	17	514

Table 2: Lines of code counted with *cloc* (`http://cloc.sourceforge.net/`).

nature of the grammar and its attribute definitions makes modifications of both fairly simple. Also, the runtime models described by a RAG can be modified easily with direct rewrite commands or with pattern-based techniques using different strategies [6]. While these properties of RAGs make them an option for the description of runtime models, the approach must also be easily usable. *RACR* provides not only a lightweight solution without any dependencies other than a running *Scheme* implementation, but also allows a compact and concise specification of runtime models. As shown in Table 2, our main programming effort went into the controller and worker programs while the model including all scheduling heuristics contains only about 500 lines of code.

Our use case shows RAG-controlled rewriting to be an adequate concept to manage a small runtime configuration. But while the size of the demo is limited due to technical constraints, for the concept to be practically useful, large systems must be treated consistently and efficiently. This is possible through *incremental evaluation*. Consistency of attribute values (and therefore scheduling heuristic) is given automatically, because whenever an attribute value is required, our approach makes sure that the value is up-to-date and consistent with the model. In general, consistency comes at a price: it threatens either the efficiency of a system (each attribute is recalculated on every call) or the simplicity (problem-specific bookkeeping is needed to determine which attributes to recompute in the event of model changes). These problems do not exist in RAG-controlled rewriting. While attribute values are cached, necessary re-computations due to model changes are triggered automatically with an attribute-dependency analysis. This enables the application of the presented approach in larger systems and also the future use of more complex and computationally expensive scheduling and adaptation algorithms. Thus, reference attribute grammars prove to be an adequate modelling formalism for cyber-physical systems.

6 Conclusion

We addressed two cyber-physical system challenges: (1) the development of well-founded, interactive and comprehensible runtime models with efficient, easily implementable analyses and (2) energy auto-tuning for energy-efficient computations. The latter requires the first; to auto tune regarding energy-efficiency

requires a runtime model that can be used to deduce beneficial reconfigurations. We proposed RAG-controlled rewriting as a solution to implement such a model and presented an indexing case study, whose challenges are representative for cyber-physical systems and energy auto-tuning. The evaluation of our solution showed energy savings of up to 17.5%. It is lightweight, extensible, declarative and efficient (comprehensible specifications yielding incremental analyses). In the future, we intend to apply our approach to larger use cases and more complex cyber-physical systems to show its scalability.

Acknowledgments: This work is supported by the German Research Foundation (DFG) in the SFB 912 "Highly Adaptive Energy-Efficient Computing".

References

1. Aßmann, U., et al.: A reference architecture and roadmap for Models@run.time systems. In: Models@run.time. LNCS, Springer (2014)
2. Baheti, R., Gill, H.: Cyber-physical systems. Impact of Control Technology (2011)
3. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer (2009)
4. Broy, M., Cengarle, M.V., Geisberger, E.: Cyber-physical systems: Imminent challenges. In: Large-Scale Complex IT Systems: Development, Operation and Management. LNCS, Springer (2012)
5. Bürger, C.: RACR: A Scheme library for reference attribute grammar controlled rewriting. Tech. Rep. TUD-Fl12-09, Technische Universität Dresden (2012)
6. Bürger, C.: Reference attribute grammar controlled graph rewriting: Motivation and overview. In: Software Language Engineering: 8th Int. Conf. ACM (2015)
7. Bürger, C., et al.: Reference attribute grammars for metamodel semantics. In: Software Language Engineering: 3rd Int. Conf. LNCS, vol. 6563. Springer (2011)
8. Eclipse Foundation: EMF, `http://www.eclipse.org/modeling/emf/`
9. Ekman, T., Hedin, G.: The JastAdd system: Modular extensible compiler construction. Science of Computer Programming (2007)
10. Hedin, G.: Reference attributed grammars. Informatica (Slovenia) (2000)
11. Lee, E.A.: Cyber physical systems: Design challenges. In: ISORC. IEEE (2008)
12. Magnusson, E., Hedin, G.: Circular reference attributed grammars: Their evaluation and applications. Science of Computer Programming 68(1) (2007)
13. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, Springer (1996)
14. Nipkow, T., Baader, F.: Term Rewriting and All That. Cambridge University Press (1999)
15. Paakki, J.: Attribute grammar paradigms: A high-level methodology in language implementation. ACM Computing Surveys 27(2) (1995)
16. Reps, T.W.: Generating Language-Based Environments. Ph.D. thesis, Cornell University (1982)
17. Rozenberg, G., et al. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1–2. World Scientific Publishing (1997/1999)
18. Söderberg, E., Hedin, G.: Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Tech. rep., Lund University (2012)
19. Steck, A., Lotz, A., Schlegel, C.: Model-driven engineering and run-time model-usage in service robotics. ACM SIGPLAN Notices (2012)
20. Watson, I., Marir, F.: Case-based reasoning: A review. The knowledge engineering review (1994)