Non-human Modelers: Challenges and Roadmap for Reusable Self-explanation

Antonio Garcia-Dominguez(✉) and Nelly Bencomo

School of Engineering and Applied Science, Aston University, Birmingham, UK
{a.garcia-dominguez,n.bencomo}@aston.ac.uk

Abstract. Increasingly, software acts as a "non-human modeler" (NHM), managing a model according to high-level goals rather than a predefined script. To foster adoption, we argue that we should treat these NHMs as members of the development team. In our GrandMDE talk, we discussed the importance of three areas: effective communication (self-explanation and problem-oriented configuration), selection, and process integration. In this extended version of the talk, we will expand on the self-explanation area, describing its background in more depth and outlining a research roadmap based on a basic case study.

1 Introduction

There is increased interest in tools managing models by themselves according to goals, rather than following a predefined script. A case study for the 2016 Transformation Tool Contest [8] on the class responsibility assignment problem showed how traditional model-to-model languages had to be orchestrated with higher-level components that guided rule applications (e.g. genetic algorithms, simulated annealing, reachability graphs or greedy application of a heuristic). Two solutions were based on reusable model optimisation frameworks (MOMoT, MDEOptimiser). These tools were evaluating options in a model much like a human would: these could be considered as "non-human modellers" (NHMs).

Self-adaptive systems based on the models@run.time approach are another good example of tools that manage models on their own. The models@run.time approach advocates using models to provide a formal description of how the system should behave. In a self-adaptive system, there is a feedback loop that guides the evolution of the model based on its current performance and the environment. Some examples of self-adaptive models@run.time include smart grid outage management [4] or ambient assisted living [9].

These NHMs are software entities which participate in a modeling team. As team members, we need to be able to influence their behaviour and understand why they took specific decisions within the models: this may be particularly complex when talking to domain experts rather than MDE specialistics. The NHMs also need to find their place in our processes and in the lifetime of the system that is being developed: they could be used once early in development, invoked once per development sprint, or brought in as another member of a

M. Seidl and S. Zschaler (Eds.): STAF 2017 Workshops, LNCS 10748, pp. 161–171, 2018.
https://doi.org/10.1007/978-3-319-74730-9_14

(possibly live) collaborative modeling tool. These challenges point to several interesting lines of research, which were raised at the GrandMDE workshop.

This extended version of the original proposal will focus on the discussion developed during the GrandMDE workshop around the self-explanation area, in which strong links with existing ideas from traceability and provenance were identified. After introducing this expanded background, a motivational case study will introduce a general roadmap for our envisioned approach. This roadmap will be grounded around existing standards and industrial-strength tools where possible.

2 Discussed Topics

Based on the previous discussion, these are some of the lines of work that we considered relevant to integrating non-human modellers (NHMs) as team members:

- Accessible configuration: existing tools have wildly different approaches to fine tune their behaviour, and domain experts find it increasingly harder to figure out which knobs to turn. It should be possible to abstract over specific approaches and provide users with a problem-centered description of any parameters, much as SPEM describes software processes abstractly.
- Self-explanation: the approaches currently available for this capability in self-adaptive systems are *ad hoc* and costly to develop, making them very rare in practice. There is no common approach for model optimisation either. This line of work would start by allowing changes to a model to be annotated with "why" a change was made: which reasoning process was followed, what inputs were used, and which other alternatives were evaluated. This would be followed up with producing accessible descriptions of this stored information.
- Selection: the community would benefit from building a coherent toolbox of options for NHMs and guiding practitioners on how to pick the right one for a particular problem, much like how the Weka tool brought together many data mining approaches into a common framework[1].
- Process integration: depending on the task, the NHM will need to be integrated into the day-to-day work of the team. Beyond one-off uses, NHMs could be part of a continuous integration loop (reacting to commits on a model), or participate in a shared modelling environment (perhaps with the ability to chat with users about observed issues).

Among these topics in our talk at the GrandMDE'17 workshop, self-explanation attracted most of the questions afterwards: attendees requested a concrete motivational example, and there were several discussions on how this challenge tied to existing work in the areas of traceability and provenance. The rest of this work will focus on answering those questions and setting out an initial roadmap for advancing the state of the art in self-explanation of NHMs.

[1] http://www.cs.waikato.ac.nz/ml/weka/.

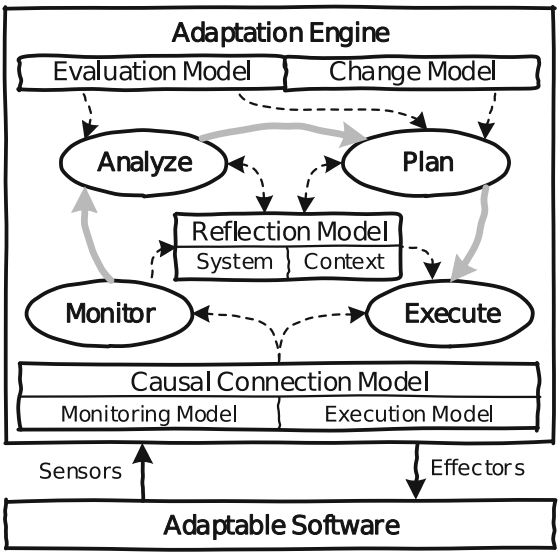

Fig. 1. MAPE feedback loop using different runtime model types [19]

3 Background for Reusable Self-explanation

Adaptive software relies on feedback loops between *sensors* that bring information from the environment, and *effectors* that change it. When it is model-based (i.e. following a models@run.time approach), these sensors and effectors will be exposed through *runtime models*. Giese et al. [10] defines runtime models as models that:

- Are designed to be used during the runtime of a system,
- are encoded to enable its processing,
- are casually connected to the system (changes in the model propagate to the system and/or the other way around).

Wätzoldt and Giese [19] presented a useful categorisation of runtime models around the MAPE (Monitor, Analyze, Plan, and Execute) loop [13], shown in Fig. 1. Within this loop, we can observe the following types of models within the adaptation engine:

- *Reflection models* reflect concerns about the running system (*system models*) and its context (*context models*).
- *Evaluation models* contain the specification of the system (*requirement models*) and any assumptions made (*assumption models*). These define the solution space of the system.
- *Change models* describe specific solutions for the space defined by the evaluation models. *Variability models* explicitly describe the available options (much like a feature model), while *modification models* only indicate what changes would be made on the reflection model.

– *Causal connection models* link the system to the reflection models. *Monitoring models* indicate how the reflection models should be updated from the system and the context, whereas *execution models* specify how to operate the system based on the information in those reflection models.

In this case, typically the evaluation and causal connection models would be produced by the developers of the adaptive system, and the reflection and change models would be managed by the adaptation engine itself. Here, the adaptation engine would be the NHM.

Self-explanation would therefore entail telling users why the adaptation engine made a specific change in those reflection and change models. This change could come from multiple reasons: changed requirements or assumptions in the evaluation models, different solution alternatives in the change models, or new data coming through the monitoring model. Alternatively, a user could simply want to see a snapshot of the reflection models at a certain moment, and request the reason why a specific value or entity was present.

Solving this problem requires combining ideas from multiple areas. So far, we have identified three: traceability, model versioning and provenance. The following sections will provide more background on each of these topics and how they relate to self-explanation for these adaptation engines.

3.1 Traceability

Linking system requirements to its various design and implementation artifacts has been a concern for a long time. Gotel and Finkelstein defined requirements traceability back in 1994 [11] as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction". Much work has been done since then in terms of defining guidelines for traceability, creating and maintaining trace links, and querying those trace links after the fact [5].

It is hard to create and maintain traceability links between manually created artefacts. However, the automation brought by MDE has made it possible to generate links between models in most of the popular model-to-model transformation languages (e.g. ATL [12] or ETL [15]). These traceability links are typically quite simple, with references to the various ends of the link (source and target elements) and the transformation rule that was applied, as mentioned by Matragkas et al. [16].

On the one hand, self-adaptive systems are automated just like model-to-model transformation engines, and therefore it should be possible to include trace creation into their processes. On the other hand, self-adaptive systems need to operate in an uncertain environment - their decisions usually require more complex reasoning and are dependent on the specific context at the time. Generic source-target links would be unable to capture this information: a more advanced information model is required. It appears that a richer case-specific traceability metamodel would be ideal for this situation, as recommended by Paige et al. [18].

3.2 Model Versioning

Whereas traceability is about following the life of a requirement or seeing where a piece of code really come from, versioning is about keeping track of how a specific artifact evolved over time. Version control systems (VCSs) have been a staple of software engineering for a very long time, and current state-of-the-art systems such as Git[2] make it possible to have developers collaborate across the globe.

Models also evolve over time, and it is possible to reuse a standard text-based VCS for it. However, text-based VCSs do not provide explicit support for comparing models across versions or merging versions developed in parallel by different developers. This has motivated the creation of specific version control systems for models: *model repositories*. EMFStore [14] and CDO[3] are mature examples. EMFStore is file-based and tracks modifications across versions of a model, grouping them into *change packages* which can be annotated with a commit message. CDO implements a pluggable storage solution (by default, an embedded H2 relational database) and provides support for branching and merging. Current efforts are focused on creating repositories that scale better with the number of collaborators (e.g. with better model merging using design space exploration [6]), or with the size of the models (e.g. Morsa [17]).

Self-explanation needs to keep track of the history of the various runtime models, and could benefit from these model repository. However, current systems only keep unstructured descriptions (in plain text) of each revision that a model goes through. It would be very hard to achieve self-explanation from these commit messages: we would rather have *commit models* that are machine readable. These commit models would have to relate the old and the new versions with external models, perhaps under their own versioning schemes. Supporting these cross-version relationships may require a good deal of research and technical work as well.

3.3 Provenance

Buneman et al. defined data provenance (also known as "lineage" or "pedigree") as the description of the origins of a piece of data and the process by which it arrived at a database [3]. This was further divided into the "why provenance" (table rows from which a certain result row was produced) and the "where provenance" (table cells from which a certain result cell was produced).

Since then, provenance has been slowly extended to cover more and more types of information systems, and has taken special importance with the advent of "big data". Commercial vendors such as Pentaho now include data lineage capabilities in their own Extract-Transform-Load tools[4].

Various efforts have been made to standardise the exchange of provenance information. In 2013, the Provenance Working Group of the World Wide Web

[2] http://git-scm.com.
[3] http://projects.eclipse.org/projects/modeling.emf.cdo.
[4] https://help.pentaho.com/Documentation/6.0/0L0/0Y0/Data_Lineage.

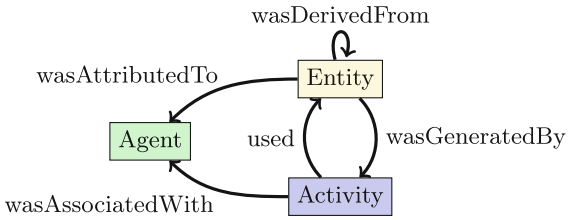

Fig. 2. High-level overview of the PROV data model [1]

Consortium (W3C) produced the PROV family of documents, which "defines a model, corresponding serializations and other supporting definitions to enable the inter-operable interchange of provenance information in heterogeneous environments such as the Web". Provenance is further generalised to knowing the origins of any digital object: provenance records indicate how entities were generated by activities undertaken by various agents [1]. This high level view of the PROV data model is shown on Fig. 2. PROV includes further provisions for specifying *roles* taken by agents and entities in an activity, and how activities may follow *plans* across time.

This view of provenance can be seen as a richer, more detailed view of traceability that not only requires following artefacts that are produced from one another, but also tracking carefully what was done and by whom. In fact, the PROV data model could be used as a starting point for the formal notation of the machine-readable "commit messages" suggested in Sect. 3.2.

4 Example Scenario

After introducing the various topics related to self-explanation for NHMs, this section will propose a concrete scenario where a reusable infrastructure for self-explanation of models@run.time approaches would be useful. It is from the service monitoring and management domain. Specifically, it is about the use of self-adaptation for the management of clusters of Hawk servers.

4.1 Scenario Description

Hawk is a heterogeneous model indexing framework [2], originally developed as part of the MONDO project. It monitors a set of locations containing collections of file-based models, and mirrors them into graph databases for faster and more efficient querying. Hawk can be deployed as an Eclipse plugin, a Java library, or a standalone server. The server version of Hawk exposes its capabilities through a web service API implemented through Apache Thrift. Prior studies have evaluated how a single Hawk server can scale with an increasing number of clients [7], with competitive or better results than other alternatives (CDO and Mogwai).

Despite these positive results, Hawk servers still have an important limitation: at the moment, there is no support for aggregating multiple servers into

a cluster with higher availability and higher scalability. Hawk can use the high-availability configurations of some backends (e.g. the OrientDB multi-master mode), but this will not improve the availability of the Hawk API itself, which will still act as a single point of failure.

The only way to solve this is to have entire Hawk servers cooperate with others: they should distribute their load evenly and monitor the availability of their peers. Periodic re-indexing tasks should also be coordinated to ensure that at least one server in the cluster will always remain available for querying, and out-of-date servers should be forced to update before becoming available again.

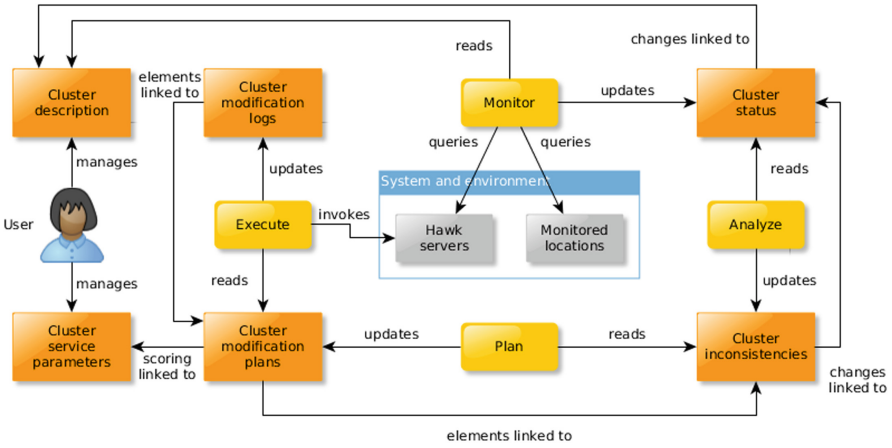

Fig. 3. Case study: self-adaptive Hawk clusters

This results in a self-adaptive system like the one shown in Fig. 3, where the adaptation layers wraps over the collection of Hawk servers (the "system") and the monitored model storage locations (the "context"):

- The "User" of the cluster manages two models: the *description model* of the intended contents of the cluster (locations to be indexed, and servers to be managed), and the *non-functional service parameters model* of the cluster (desired tradeoff between availability and freshness).
- The "Monitor" step of the MAPE loop uses the description as a monitoring model and queries the various servers and locations. The obtained information is used to update the *cluster status model*, a reflection model where each change is annotated with metadata (e.g. query timestamp, response time, observed errors) and linked back to the element of the cluster description model that caused it.
- The "Analyze" step takes the cluster status model and revises the *cluster inconsistencies* model, removing previously observed inconsistencies that no longer hold (linking them back to the evidence) and adding new inconsistencies (again, linked to the evidence) where servers are outdated or unavailable.

The cluster inconsistencies model would be a change model, as it reflects the problem to be solved in the "Plan" step.
- The "Plan" step produces a set of *cluster modification plan models* with different alternatives to solve the inconsistencies. These alternatives are different sequences of invocations of the Hawk management API. Their elements may solve (partially or fully) certain inconsistencies, and each plan will need to be scored according to the cluster service parameters. These could be considered as execution models.
- Finally, the "Execute" step will run the highest scoring modification plan and record the results of those invocations in a *modification log model* (yet another reflection model). Each log entry needs to be justified from the original elements of the selected modification plan.

4.2 Approach for Reusable Self-explanation

Each of these MAPE steps is acting as its own NHM, taking in the latest versions of independently evolving models and combining them with external information to drive the evolution of its own runtime models. Allowing users to develop trust by gaining an understanding of the workings of the adaptation engine is important for the wider adoption of Hawk clustering. It may seem that tracking all these adaptations would require purpose-specific models and infrastructure, but on closer examination, there are already many elements in the state of the art that can be reused.

The PROV Data Model is a promising start for a reusable metamodel for self-explanation of changes in runtime models (Sect. 3.3). However, there are multiple ways in which PROV could be used:

1. PROV explains each element in each version of the runtime models. The activities are the MAPE steps, the agent is the adaptation engine, and the entities are the source pieces of evidence for the element. It is not exactly clear how would the reasoning for a particular element being there would be linked.
2. PROV explains each change applied to each version of the runtime models. The activities are the changes themselves, the agent is the MAPE step, and the entities are the source pieces of evidence for the element, as well as the reasoning for that change (or group of changes). Our initial estimation is that this approach would provide more fine-grained information for later self-explanation.

Regardless of our selection, PROV lacks specific provisions for model management activities, or pointing to certain model elements within a specific version of a model. These would need to be defined and developed.

Tracking the history of the models themselves can already be done through the model repositories mentioned in Sect. 3.2, but it is unclear how to store the above PROV descriptions and link them back to the various revisions of the models. The PROV descriptions could be kept as complementary models in the

repository, or they could be kept as *commit models* which replace the usual textual commit messages. Using complementary models could be easier to reuse across model repositories, but some conventions would still be needed to link the tracked model to the PROV information. Using commit models could be more natural if the model repository already tracked model changes in its commits (like EMFStore): links could be directly established to those elements. This is another area that merits further research.

Finally, once the PROV records have been created and stored, the next part would be presenting them in a manner that is approachable but still reusable across applications. Conceptually, what we want is (i) being able to track the state of the MAPE loop at a certain point of time, and more importantly (ii) answering how a model element or value in it came to be. While (i) should be readily available through sensible use of model versioning, we are not aware of a reusable interface for querying the information in (ii). Conceptually, it would be a more powerful version of the capability of most version control systems to know when was a certain line of text touched, and by whom.

5 Research Roadmap

Summarising the discussion from the previous section, we can establish an initial roadmap for the creation of a first prototype of the envisioned reusable self-explanation framework for self-adaptive systems following the MAPE loop:

1. First of all, the creation of *basic self-adaptive prototypes* that achieve the intended functionality except for the self-explanation capabilities.
   Deriving the self-explanation framework from working software (bottom up) should provide a more realistic implementation, and will give us more experience in the implementation of self-adaptive systems according to the models@run.time paradigm. Hawk will benefit earlier from the horizontal scalability as well.
2. Next, the prototype would be extended with *simulation capabilities* for the system and environment, in order to create new situations for testing adaptability more quickly. This would make it possible to see exactly how the system adapted to a predefined situation, and check if the self-explanation capabilities meet our expectations.
   Ideally, because of the models@run.time approach, this should be a matter of mocking the answers from the system - the rest of the approach should remain as is.
3. Being able to run the self-adaptive system in real and simulated environments, the next part of the work would be *comparing* the two approaches to extending PROV that were observed. Initially, it would be a matter of recording the information in both ways (element-first or change-first) and comparing their level of detail, and the relative ease of capture and querying.
4. This would be followed by the *integration* of the PROV records into the history of the runtime models, whether as side-by-side models or as the previously mentioned *commit models*. The comparison would also need to take into

account practical details such as scalability over time and model complexity, ease of use and reusability across different model repositories.
5. Finally, the *visualisation* of the PROV records could be treated in multiple ways. The envisioned goal is for a side-by-side view of the selected model element and its history, where the user may be able to pull additional PROV-encoded information on demand, at least over a single iteration of the full MAPE loop. The links identified between the models in Fig. 3 would enable this, unless proven otherwise during the design of the PROV extensions.

6 Conclusion

This paper started from a general proposal for the thorough consideration of goal- or requirement-based non-human entities managing models (the so-called "non-human modellers") as additional members of a modelling team that we must talk to, understand, pick and integate into our processes. The "reusable self-explanation" part took the most questions during the event, and for that reason we expanded on some of the background behind these ideas and described a scenario from the service monitoring domain in which it would be useful.

The discussion has touched upon the fact that most of the ingredients already exist: traceability and provenance have been around for a long time, and model versioning is a common practice in industrial MDE environments, with mature purpose-specific software to do it. However, our understanding is that their specific combination for reusable self-explanation has yet to be achieved, and for that purpose we have set out a bottom-up roadmap which starts with the development of a testbed and continues with the extension, storage and reusable visualisation of a dialect of the PROV specification.

References

1. PROV Model Primer. W3C Working Group Note, World Wide Web Consortium, April 2013. https://www.w3.org/TR/2013/NOTE-prov-primer-20130430/#intuitive-overview-of-prov
2. Barmpis, K., Kolovos, D.: Hawk: towards a scalable model indexing architecture. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary. ACM (2013). http://dl.acm.org/citation.cfm?id=2487771
3. Buneman, P., Khanna, S., Tan, W.-C.: Why and where: a characterization of data provenance. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 316–330. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44503-X_20. http://dl.acm.org/citation.cfm?id=645504.656274
4. Burger, E., Mittelbach, V., Koziolek, A.: View-based and model-driven outage management for the smart grid. In: Proceedings of the 11th International Workshop on Models@run.time, Saint Malo, France, pp. 1–8. CEUR-WS.org, October 2016. http://ceur-ws.org/Vol-1742/MRT16_paper_1.pdf
5. Cleland-Huang, J., Gotel, O.C.Z., Huffman Hayes, J., Mäder, P., Zisman, A.: Software traceability: trends and future directions. In: Proceedings of the on Future of Software Engineering, FOSE 2014, pp. 55–69. ACM, New York (2014). http://doi.acm.org/10.1145/2593882.2593891

6. Debreceni, C., Ráth, I., Varró, D., De Carlos, X., Mendialdua, X., Trujillo, S.: Automated model merge by design space exploration. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 104–121. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_7
7. Garcia-Dominguez, A., Barmpis, K., Kolovos, D.S., Wei, R., Paige, R.F.: Stress-testing remote model querying APIs for relational and graph-based stores. Softw. Syst. Model. 1–29 (2017). https://link.springer.com/article/10.1007/s10270-017-0606-9
8. Garcia-Dominguez, A., Krikava, F., Rose, L.M. (eds.): Proceedings of the 9th Transformation Tool Contest, CEUR Workshop Proceedings, vol. 1758, December 2016. http://ceur-ws.org/Vol-1758/. ISSN 1613–0073
9. Garcia Paucar, L.H., Bencomo, N.: Runtime models based on dynamic decision networks: enhancing the decision-making in the domain of ambient assisted living applications. In: Proceedings of the 11th International Workshop on Models@run.time, Saint Malo, France, pp. 9–17. CEUR-WS.org, October 2016. http://eprints.aston.ac.uk/29790/
10. Giese, H., Bencomo, N., Pasquale, L., Ramirez, A.J., Inverardi, P., Wätzoldt, S., Clarke, S.: Living with uncertainty in the age of runtime models. In: Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.) Models@run.time. LNCS, vol. 8378, pp. 47–100. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_3
11. Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Proceedings of IEEE International Conference on Requirements Engineering, pp. 94–101, April 1994
12. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2006, pp. 719–720. ACM, New York (2006). http://doi.acm.org/10.1145/1176617.1176691
13. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). http://ieeexplore.ieee.org/abstract/document/1160055/
14. Koegel, M., Helming, J.: EMFStore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, pp. 307–308. ACM (2010)
15. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69927-9_4
16. Matragkas, N.D., Kolovos, D.S., Paige, R.F., Zolotas, A.: A traceability-driven approach to model transformation testing. In: Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, 29 September 2013 (2013). http://ceur-ws.org/Vol-1077/amt13_submission_7.pdf
17. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: A repository for scalable model management. Softw. Syst. Model. **14**(1), 219–239 (2015). https://link.springer.com/article/10.1007/s10270-013-0326-8
18. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous identification and encoding of trace-links in model-driven engineering. Softw. Syst. Model. **10**(4), 469–487 (2011). http://link.springer.com/10.1007/s10270-010-0158-8
19. Wätzoldt, S., Giese, H.: Classifying distributed self-* systems based on runtime models and their coupling. In: Proceedings of the 9th International Workshop on Models at run.time, pp. 11–20 (2014). http://st.inf.tu-dresden.de/MRT14/papers/mrt14_submission_3.pdf