An Infrastructure for Generating Run-time Model Traces for Maintenance Tasks

Lorena Arcega*†, Jaime Font*†, Øystein Haugen‡ and Carlos Cetina*
*Universidad San Jorge, SVIT Research Group, Zaragoza, Spain
Email: {larcega,jfont,ccetina}@usj.es
†University of Oslo, Department of Informatics, Oslo, Norway
‡Østfold University College, Department of Information Technology, Halden, Norway
Email: oystein.haugen@hiof.no

***Abstract*—Current research efforts are focused on taking advantage of the models at run-time for run-time decision-making related to run-time system concerns associated with autonomic and adaptive systems. In addition, all systems need maintenance over time as new requirements emerge or when bug-fixing becomes necessary. Models at run-time can provide an execution trace of a high level of abstraction that is useful for maintenance tasks. In this paper, we propose a generic infrastructure, which is able to get the run-time model trace. Our infrastructure creates a descriptive model of the running code by means of Code-Model Connection Rules. These rules translate the behaviour of the running source code in model traces. We validate our infrastructure in a Smart Hotel. The results of our infrastructure show promising results towards the generation of model traces from source code at run-time. However, further work is required to a better specification of the rules, to solve some issues with the model dependencies and to allow the propagation of changes in the models to the source code.**

I. Introduction

Models at run-time research efforts are concerned with how abstractions of software implementations can be used at run-time to manage software changes at run-time [1]. Current research works are focused on how models can be used to support run-time adaptations in autonomous systems (i.e., in self-* systems) [2]. Usually, the adaptation actions are driven by a MAPE-K loop [3] and prescriptive models. That is, changes in the model are transferred to the system.

In software development, all systems evolve over time as new requirements emerge or when bug-fixing becomes necessary. Lehman et al. [4] pointed out that up to the 80% of the lifetime of a system is spent on maintenance and evolution activities. Currently, research efforts in feature location are concerned with identifying software artifacts associated with a program functionality (a feature). Feature location is one of the most important and common activities performed by developers during software maintenance and evolution [5].

The amount of run-time data produced by many applications tends to be huge and is recorded in different forms. Relevant information has to be extracted and stored with the corresponding artifacts produced at design-time to reveal the causes of unexpected software behavior [6]. The purpose of the analysis is to find the effects within a chain of effects occurred during software execution [7]. Therefore, some techniques are needed to extract and understand the run-time information. Then, run-time information is necessary at design-time to make decision regarding the system maintenance.

The goal of our work is to extend the use of models at run-time to perform maintenance task. This work presents our generic infrastructure to generate run-time model traces through the use of connection rules specified at design-time called *Code-Model Connection Rules*.

The *Code-Model Connection Rules* maps model elements to different point in the source code such as methods or values of variables. Our generic infrastructure is divided in two parts: Controller and Translator. It controls the implementation code, check it through the *Code-Model Connection Rules*, and, if changes in the run-time model are necessary, our generator translates the changes in the model at run-time.

We validate our infrastructure in our Smart Hotel. The infrastructure creates model traces from source-code to models. The results of the validation of our infrastructure show promising results towards the generation of model traces from source code at run-time. However, further work is required to a better specification of the *Code-Model Connection Rules*, to solve some issues with the model dependencies and to allow the performance of maintenance tasks at model level, that is, the propagation of changes in the models to the source code.

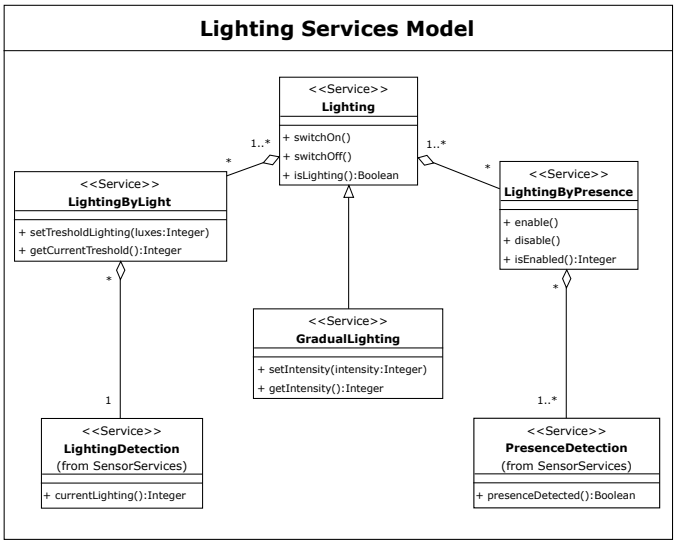
The remainder of the paper is structured as follows. In Section II, we introduce the Smart Hotel. In Section III, we describe the steps of our approach. In Section IV, we validate our approach with the Smart Hotel case study and then discuss the results. In Section VI, we examine the works related with this one. Finally, in Section VII, the conclusions of our work are presented.

II. Background

The running example and the evaluation of this paper are performed through a Smart Hotel [8]. In this section, we present the models that are used by the Smart Hotel.

We use Pervasive Modeling Language (PervML) [9] to describe the Smart Hotel. PervML[1] is a Domain Specific Language (DSL) that describes pervasive systems using high-level abstraction concepts such as services. This language

[1]https://tatami.dsic.upv.es/pervml/index.php



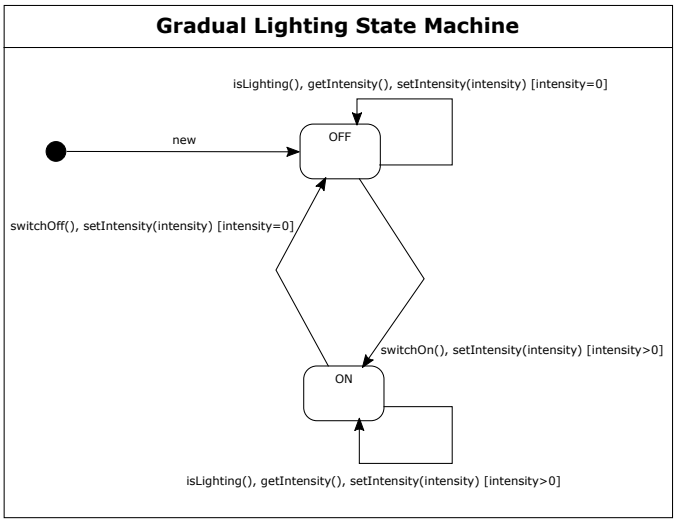


Fig. 1. Smart Hotel PervML Models

is focused on specifying heterogeneous services in distinct physical environments. This DSL has been applied to develop solutions in the Smart Hotel domain.

PervML uses six different models which globally provide the information that is required to describe a pervasive system. These models are:

- The **Services Model** describes the kinds of services that are provided in the pervasive system. It is used for specifying the common characteristics of these similar services.
- The **Structural Model** indicates the number and kind of services that are provided by the pervasive system in every system location.
- The **Interaction Model** is used for describing the communication that is produced between the different components of the services.
- The **Binding Providers Model** is used to describe the different kinds of devices or software systems that are in charge of providing the basic pervasive system functionality.
- The **Components Structural Specification** is used to assign devices and software systems to each one of the service components.
- The **Components Functional Specification** is used to specify the actions that are executed when an operation of a service is invoked.

Because of space constraints, Fig. 1 shows only a small part of the PervML models related to the functioning of the lights in the Smart Hotel. The concrete syntax of PervML is similar to the one used by UML. The Smart Hotel provides services which offer operations for switching on and switching off the lights or setting the light intensity. The Gradual Lighting State Machine shows the state machine related to the gradual lights. This kind of service supports setting the light intensity from 0% to 100% by means of a set of operations.

The output system is obtained through a model to text transformation. That is, each element of the model is translated to one or more Java bundles. In addition, we use an implementation framework of PervML and the OSGI framework [10]. See [9] for more implementation details.

III. From Source-Code to Models

This work presents our generic infrastructure to generate run-time model traces through the use of connection rules specified at design-time called *Code-Model Connection Rules*. The infrastructure monitors the software changes in the running system and put the running information into a descriptive model at run-time.

Fig. 2 shows an overview of our Run-time Model Trace Generator. At design-time, the software engineer creates the models that specify the system and are used to generate the run-time model. From the models, the code is implemented, usually manually by using the models as a reference and in some cases it is doing automatically or semi-automatically. The software engineer also species the rules, *Code-Model Connection Rules*, which will generate the descriptive model at run-time.

In a regular use, the deployment of the system is performed running the source code. When the software engineer wants to obtain the run-time model trace, he can activate the Run-time Model Trace Generator.

In the Run-time Model Trace Generator, our infrastructure controls the implementation code, checks it through the *Code-Model Connection Rules*, and, if changes in the run-time model are necessary, our generator translates the changes in the model trace at run-time.

As a summary, the key idea of our infrastructure is a loop which is responsible for translating the information of the run-time knowledge from source code to models. Next subsections present the *Code-Model Connection Rules* used in our infrastructure and the steps of our Run-time Model Trace Generator.

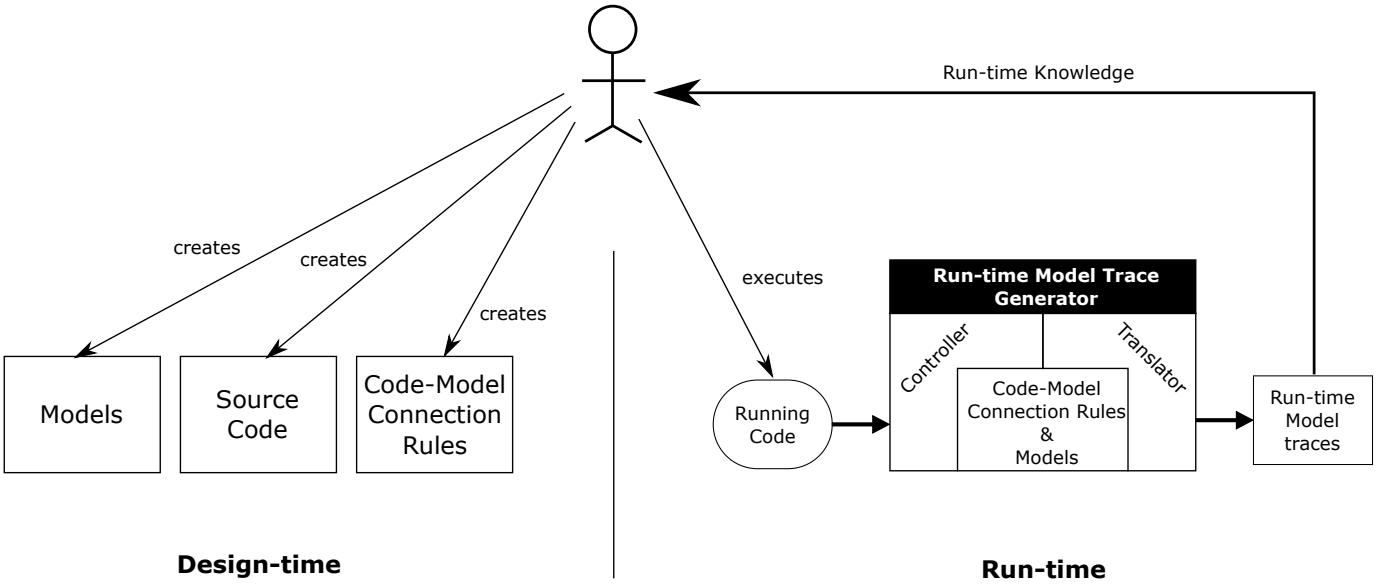


Fig. 2. Overview of the Run-time Model Trace Generator

A. The Code-Model Connection Rules

To generate a model from the executed source code, we developed a *Code-Model Connection Rules*, which are expressed using Event-Condition-Action (ECA) rules [11]. We are not reinventing the ECA rules. The ECA rules have been intensely studied and used for the management of self-adaptive systems [8]. However, we use these ECA rules in a novel way to perform changes in a run-time model when the code is executed.

The definition of our rules is based on the ERA language [12] to provide mechanism for specifying the execution of external actions (actions in the run-time model) instead of internal actions (actions in the running code of the system).

The *Code-Model Connection Rules* take the following shape: *'when a line of the source code is executed, the state of variables change, or both take place, then a model element should be created, modified, or deleted in the run-time model'*.

For example, when the line of code that creates the gradual lighting service *(GradualLightService mainGradualLightService=new GradualLightService())* is executed, the GradualLighting model element in the run-time model must appear. In this particular case, the rule is depicted as follows:

```
(GradualLightService mainGradualLightService=new
    GradualLightService(),
    IS_NOT(LightingServices.GradualLighting),
    ADD_ELEMENT(LightingServices.GradualLighting));
```

In this rule, the first parameter (*GradualLightService mainGradualLightService=new GradualLightService()*) is the line of code that triggers the rule. The second parameter (*IS_NOT(LightingServices.GradualLighting)*) indicates the condition that must be fulfilled and in which model. The keyword *IS_NOT* checks if the model element appears in the run-time model and return true if it is not in the model. In this case it returns true if the model element GradualLighting is not in the LightingServices model. The third parameter (*ADD_ELEMENT(Lighting- Services.GradualLighting)*) is the kind of modification that must be performed in the run-time model. In the smart hotel, it is the addition of the model element GradualLighting in the LightingServices model.

On the other hand, when the code indicates that the GradualLightingService is not available any longer (*remove(mainGradualLightService)*), the element must be removed from the run-time model. The rule is similar to the one for adding model elements:

```
(remove(mainGradualLightService),
    IS( LightingServices .GradualLighting),
    DEL_ELEMENT(LightingServices.GradualLighting));
```

The first parameter (*remove(mainGradualLightService)*) is the line of code that must be executed to trigger the Model-Code Connection rule. The second parameter *(IS(LightingServices.GradualLighting))* indicates if the model element is in the model. The third parameter (*DEL_ELEMENT(LightingServices.Gradual- Lighting)*) is the modification that must be performed in the run-time model. In this case it is a deletion of the model element GradualLighting in the LightingServices model.

Although the idea of the *Code-Model Connection Rules* is domain independent, our current implementation is tailored to the Smart Hotel. Even though these rules have helped us to address the research questions, our future work involves designing independent domain rules that combine debug expressiveness to specify the state of the variables and execution points (triggers) and OCL-like expressions to specify (conditions and actions in the run-time model).

Some model elements may require few rules to be synchronized with the code, but others may require more effort.

These rules are a trade-off, the more specifications the engineer makes, the richer the run-time model will be, but at the cost of the increase in the amount of time that must be spent by the software engineer in order to specify them.

One of the main characteristics of our approach is that, by means of the *Code-Model Connection Rules*, the software engineer can change the set of the models defined at design-time that will be used at run-time. That is, the software engineer can decide how many model elements should be connected with the running code.

B. Run-time Model Trace Generator

Szvetis and Zdum [13] creates a classification of the objectives, techniques, kinds and architectures used with models at run-time. According to their categories, we can classify our infrastructure to work with models at run-time as follows:

- Objectives: The main one is monitoring the system by using models in order to help the understanding of the system behavior. In addition, we use models to raise the abstraction level and bring closer the problem to the user. Finally, the use of models provides platform-independent views of the system under observation.
- Technique: We use the introspection technique that deals with extraction of run-time system data in order to apply model-based techniques on the analyzed behavior. In addition, they subdivided the introspection in three main groups: event log checking, instrumentation and management API.
  Specifically, our infrastructure is similar to the instrumentation group. Even though, we do not insert any extra source code to the system. We use the rules to translate the running software of a system into a model.
- Kind: Our infrastructure does not depend on one kind of model. We use a domain specific language (PervML, see section II), however, the model generated depends on the defined rules and the user can choose whoever he wants.
- Architecture: We use a model-aware middle-ware architecture. This architecture allows us to monitor the system while our solution checks the rules and creates the model at run-time.

The Code-Model Connection rules previously presented enables to reflect running-code changes into a descriptive run-time model. The details of our infrastructure are depicted in Fig. 3.

- **Controller:** When the system is running, the running source code needs to be controlled (see Fig. 3). The implementation of the controller is developed by means of the Java Code Language Introspection Capabilities [14].
  Our infrastructure uses the run-time state of the code as input to check conditions from the *Code-Model Connection Rules*. In the running example, our approach detects every time that the line *'GradualLightService mainGradualLightService = new GradualLightService()'* is executed.
  Once a relevant change in the state of the code is detected, our approach checks if the conditions of the *Code-Model Connection Rule* is fulfilled. In the running example, our approach checks if the GradualLighting model element appears in the LightingServices run-time model, *('IS_NOT(LightingServices.GradualLighting)'*).
- **Translator:** The response of the controller is used to translate the source code into models (see Fig. 3). The implementation of the translator is developed by means of the EMF Introspection Capabilities [15].
  Our infrastructure uses the action of the *Code-Model Connection Rule* to determine the modifications that must be performed in the run-time model. In the running example, the action determines that the GradualLighting model element must be added to the LightingServices run-time model (*'ADD_ELEMENT(LightingServices.GradualLighting)'*). If the model element does not appear in the run-time model, it checks the design-time models and takes the information that is needed in order to add this model element to the run-time model. If the model element appears in the run-time model, our approach checks if some modifications are necessary.
  The run-time model is modified in order to add, modify or delete the model elements specified in the *Code-Model Connection Rules*. In the running example, the GradualLighting model element is added to the run-time model.

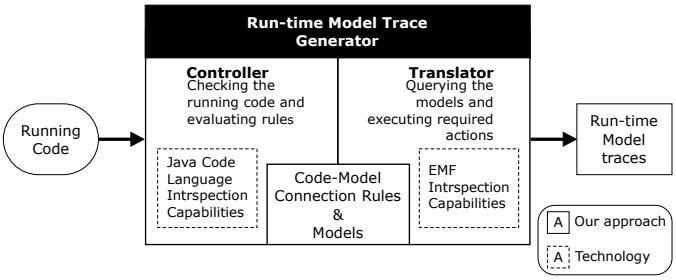


Fig. 3. Details of the Run-time Model Trace Generator

The model generated at run-time only includes elements that have been executed. This run-time model is the main artifact that the software engineer can use to improve his knowledge of the system. We extract the run-time representation of the system as models to enable reasoning on a high level of abstraction.

IV. Validation

We validate whether our infrastructure creates the correct models at run-time. To do that, we compare our model traces generated at run-time with an oracle that is composed by the models of the defined scenarios in [8]. We generate a run-time model trace while the other model trace is extracted from a previous work [8], we compare both traces in order to detect differences.

We defined the experimental design of our study using the Goal-Question-Metric method (GQM) [16]. We used the template presented in [17]. The GQM method was defined as a mechanism for defining and interpreting a set of operation

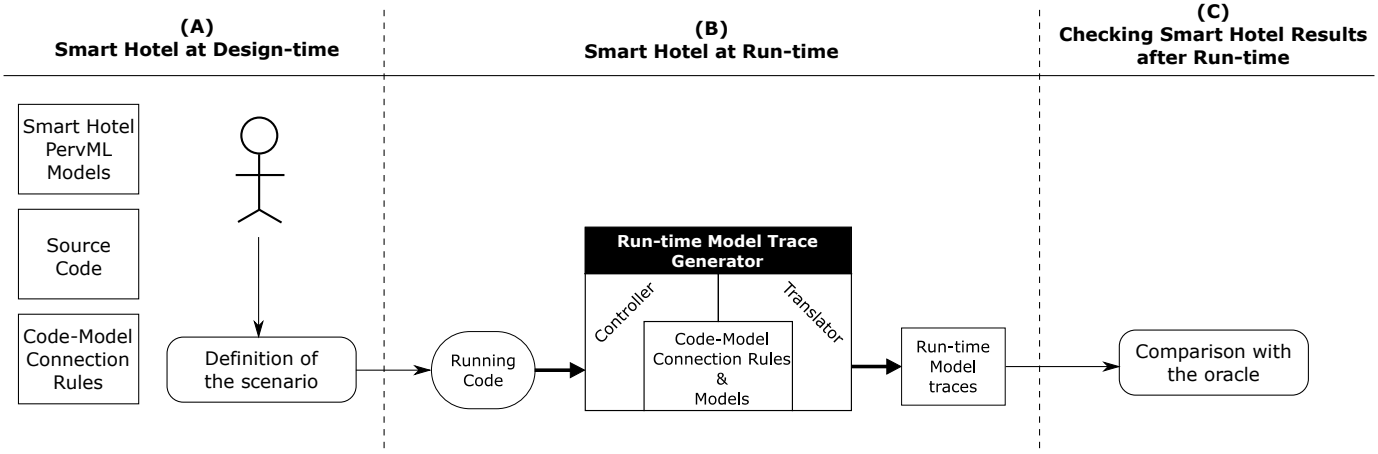


Fig. 4. The validation process followed in the Smart Hotel

goals using measurements. In this evaluation, our goal was the following:

- Object: Our Smart Hotel
- Purpose: Validation
- Issue: The correctness of the models generated at run-time
- Context: The infrastructure for of our Run-time Model Trace Generator

To fulfill this goal, we focused on answering the following research question:

1) Does our infrastructure generate the same model traces as the oracle?

Basili in [16] and Travassos in [18] describe four kinds of studies: in-vivo, in-vitro, in-virtuo, and in-silico. In our case, we chose to carry out in-virtuo experiments, where the real world is described as computer models. This experiment involves the interaction among participants and a computerized model of reality. The simulated environment offers major advantages regarding the cost and the feasibility of replicating a real-world configuration. In addition, some scenarios such as res or floods cannot be replicated in the real world.

The Smart Hotel is developed by means of a software product line, the PervML model and the source code of the Smart Hotel are configured with a feature model. The feature model specifies the 39 different features that the Smart Hotel has implemented. We use the PervML model traces of the [8] approach as an oracle to evaluate the presented approach. That is, we make use of a set of implementation codes whose model realizations are known beforehand and will be considered as the ground truth, enabling us to compare the run-time traces extracted from the [8] approach with the results of the models generated by our infrastructure.

Figure 4 shows the entire process that we have followed in this validation. For the validation, we used the Smart Hotel defined in [8]. The Smart Hotel presents 476 model elements in the design-time models. It consists of 268 classes implemented in about 67,207 methods of Java source code. In the evaluation set-up, a simulated environment was used to represent the Smart Hotel. In addition, at design-time, the *Code-Model Connection Rules* and the scenario are defined to allow the generation of the trace model from the code at run-time, as shown in Fig. 4 (A).

After running the scenario, see Fig. 4 (B), our Run-time Model Trace Generator (see Fig. 4 left) generates the run-time model. The infrastructure monitors the running code and applies the corresponding *Code-Model Connection Rules* to translate source code behavior into a descriptive run-time model trace.

For the comparison, see Fig. 4 (C), we collect the traces by means of the Reconfiguration Tracker that is provided by the implementation of the Smart Hotel that we used [8].

At the end, we have two set of models that represents the behavior of the system. One of the set is generated by our Run-time Model Generation and the second set is the oracle. Then, we can compare the results of the two approaches.

The number of model elements in the run-time model depends on the *Code-Model Connection Rules* defined. As we wanted to determine if the models generated was equivalent, we performed the validation with a set of rules that connected all the possible elements with the source code. By this way, we could compare both models traces easily.

Therefore, the number of rules that the software engineer must use is an open topic that will be addressed in the future. The specification of the rules takes times and it is not always necessary to specify all of them. Thus, I hope I can guide the software engineer about what are the rules he need to specify.

A concrete example of our validation is shown in Fig. 5. The Smart Hotel is defined with a large number of models (see section II), however in this section we only show one of these models. As the examples show, all the models obtained are a subset of the entire definition of the Smart Hotel.

The fist one of the models of Fig. 5, Smart Hotel PervML Structural Model, corresponds to the one that defines the entire system of the Smart Hotel. The second one, Smart Hotel PervML Structural Model in Watching a Movie Scenario represents the model generated in the Watching a Movie

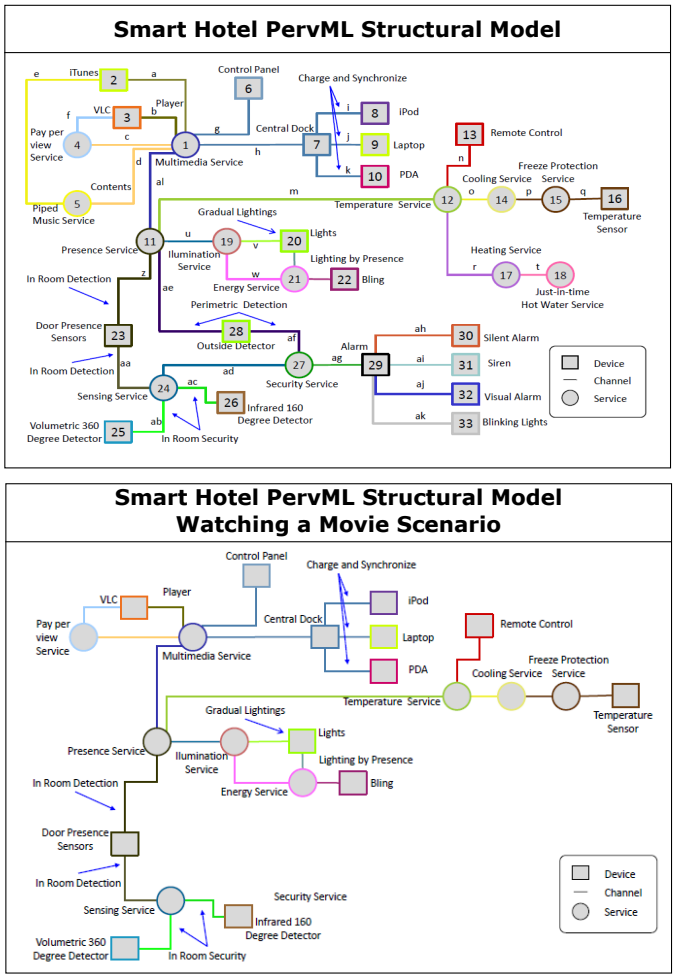


Fig. 5. Smart Hotel PervML Structural Models

Scenario.

The definition of the Watching a Movie scenario is as follows:

'The user of the hotel decides to watch a movie from the of which the hotel offers. The user can select the movie in the screen of the control panel. In addition, the user will be able to choose the room preferences (illumination and temperature) for a better experience. The system will give him many options so he can choose the one that suits him more. The user is enjoying the movie but, someone is calling to his cell phone with Bluetooth technology synchronized with the rooms main system. This call will be notified through the main screen. He is forced to stop watching the movie but he will have the option to continue where he left at any other time during his stay.'

Then, the main devices that are involved in this scenario are the cell phone, the control panel, the central dock. Some other devices related to this scenario are ones related to the illumination and temperature services. In the same way, the main service in this scenario is the multimedia service, and also the ones related to illumination and temperature.

This trace generated from this scenario is formed by 23 channels, 13 devices and 9 services in the Smart Hotel PervML Structural model (see bottom part of Fig. 5).

A. Does our infrastructure generate the same model traces as the oracle?

The results of our infrastructure show promising results towards the generation of model traces from source code at run-time.

The results obtained in this validation suggest that our infrastructure generates a valid model trace from the running scenarios. All the generated traces contain the main devices and services necessary for execute the defined scenario.

However, the trace generated with our infrastructure contained more information than the trace from the oracle. But, the trace from the oracle was contained in the trace generated with our infrastructure. The major difference between our generated model traces at run-time and the traces from the oracle lies in the number of attributes of the model element.

In addition, it is necessary further specific experiments in order to identify more relevant correlations between the source code and the run-time models. However, the *Code-Model Connection Rules* have flexibility to enable and disable the model elements that must appear in the run-time model trace.

Another issue detected is related to the dependencies between model elements. The management of the dependencies through the use of the *Code-Model Connection Rules* is cumbersome and error prone. We detect that a dependency tree could be very useful to check the consistency of the model trace while it is being generated.

Moreover, our infrastructure generates the model trace to increase the level of abstraction and to ease the maintenance task. But nevertheless, the maintenance task must be performed at source code level because the infrastructure does not allow propagate changes in the models to the source code of the system.

In spite of the results, this initial effort must not be underestimated since the Run-time Model Trace Generator produced proper output to see the first results. The overall performance of the infrastructure needs more tests. Further work is required to see if these concerns are reasonable and if so, find out how to these previous problems can be solved.

V. Threats to Validity

We use the classification of threats to validity in [19]. This classification distinguishes four aspects of validity:

1) **Construct validity:** This aspect of validity reflects the extent to which the operational measures that are studied represent what the researchers have in mind and what is investigated based on the research questions.
   The purpose validation does not have a true/false answer. To minimize this threat the scenarios were designed by two experts in the Smart Hotel. These experts have developed the Smart Hotel and in other model-based tools (in the induction hob domain and train control software domain). Their participation was limited to the design of the scenarios.

The measure used in our research is the similarity of our model traces extracted and the oracle. These similarity measures have been proved in other kind of models with fair results [20].

2) **Internal validity:** This aspect of validity is of concern when causal relations are examined. There is a risk that the factor being investigated may be affected by other neglected factors.
   In our work, this is a very low risk due to the human task are reduced to the minimum. The comparison between the traces were performed automatically by a computer, therefore the humans did not affect the final result.
3) **External validity:**This aspect of validity is concerned with to what extent it is possible to generalize the finding, and to what extent the findings are of relevance for other cases.
   The software system used in the validation is representative of those used in practice. Given the scale and complexity of our Smart Hotel, we consider our evaluation a good starting point representing a realistic case. However, this threat can be reduced if we experiment with other software systems of different sizes and domains.
4) **Reliability:** This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers.
   The main risk relies on the selection of the running scenarios to obtain the execution trace of the models. Since we are experts in the Smart Hotel system, we can claim that our scenarios are good representatives of the entire system, and, therefore, our infrastructure generates the correct models for all the system. However, depending on the chosen scenarios, the result may differ.

VI. Related Work

Some approaches use the run-time models to visualize and analyze data from model-based systems. Although they do not generate the run-time models, their works are based on extracting information from run-time models.

Graf and Mller-Glaser [21] define various debugging-perspectives independent of the actual execution platform for identifying error occurrences. They extract run-time information out of the executed binary from the target platform and transport this information back to the model level. Obtained run-time information can then be used to visualize the internal state of the executable. In contrast to our approach, they extend the UML meta-model with meta-classes that allow storage of data acquired by the model mapping.

Maoz [22] introduces an approach which uses model-based traces to record the run-time of a system through abstractions provided by models used in its design. The traces include information about the systems from which they were generated, the models that induced them and the relationships between them at run-time. The approach is focused on metrics and operators for such traces to understand the structure and behavior of a running system while we are focused on the infrastructure to generate the run-time model from source code.

Bodenstaff et al [23] uses event log checking to check a model against the running system. They assume that the event log is consistent with the running system and also with a model if essential parts of the model do not contradict. In contrast to our approach, they are focus on identifying relevant parts of the log event and extract them as models while our approach generates the models directly from the running system.

Szvetits et al. [6] propose the usage of run-time models in combination with model-driven techniques and interactive visualizations to ease tracing between log file entries and corresponding software artifacts. They creates a repository-based approach to augment root cause analysis with interactive tracing views to maximize the re-usability of models created during the software development process. In contrast to our approach, they focus their work on the interactive visualizations while our work is focused on the generation of run-time models from non-model-based system. However, we can apply their techniques to analyze the model traces that we extract from the running system.

Some other approaches insert monitoring functionality into the system in order to observe the system behavior.

Nierstrasz et al. [24] use instrumentation at run-time in combination with and Abstract Syntax Tree (AST) model as abstraction above the byte-code level. They use the concept of links to be set as annotations to AST nodes which are transferred by a compiler plugin before execution. This results in code to be inserted in the program at nodes where links are installed. Such links can also be installed by other links or manipulated by themselves.

Hamann et al. [25] propose monitoring of JVM hosted applications by using platform-aligned model (PAM) as link between the platform-specific model (PSM) and the platform-independent model (PIM). A PAM is iteratively designed through assumptions which are stated, checked, and refined.

This kind of approaches requires the source code of the system to be available. In contrast to us, this approaches extract the run-time data and then they translate the extracted data as models to enable reasoning on a high level of abstraction while we extract the data as models at run-time and do not introduce anything new in the source code.

VII. Conclusion

This work proposes a generic infrastructure to generate run-time models through the use of connection rules called *Code-Model Connection Rules*. Specifically, our infrastructure allows change from source-code to descriptive models increasing the abstraction level.

Our validation in our Smart Hotel compares the result obtained from our infrastructure and from a oracle. We use the model traces generated to determine if the results obtained are correct. The results of the validation of our infrastructure show promising results towards the generation of model traces from source code at run-time. However, further work is required to a better specification of the *Code-Model Connection Rules*, to

solve some issues with the model dependencies and to allow the performance of maintenance tasks at model level, that is, the propagation of changes in the models to the source code.

In addition to the previous one, our future work involves designing independent domain rules that combine debug expressiveness to specify the state of the variables and execution points (triggers) and OCLlike expressions to specify (conditions and actions in the run-time model).

ACKNOWLEDGMENT

This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the project Model-Driven Variability Extraction for Software Product Line Adoption (TIN2015-64397-R).

REFERENCES

[1] U. Aßmann, N. Bencomo, B. H. C. Cheng, and R. B. France, "Models@run.time (Dagstuhl Seminar 11481)," *Dagstuhl Reports*, vol. 1, no. 11, pp. 91–123, 2012. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2012/3379

[2] H. Giese, N. Bencomo, L. Pasquale, A. J. Ramirez, P. Inverardi, S. Wätzoldt, and S. Clarke, *Models@run.time: Foundations, Applications, and Roadmaps*. Cham: Springer International Publishing, 2014, ch. Living with Uncertainty in the Age of Runtime Models, pp. 47–100.

[3] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2006.

[4] M. M. Lehman, J. Ramil, and G. Kahen, "A paradigm for the behavioural modelling of software processes using system dynamics," Imperial College of Science, Technology and Medicine, Department of Computing, Tech. Rep., Sep 2001.

[5] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," in *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[6] M. Szvetits and U. Zdun, "Enhancing root cause analysis with runtime models and interactive visualizations," in *8th International Workshop on Models at Run.time*, Miami, Florida, USA, September 2013. [Online]. Available: http://eprints.cs.univie.ac.at/3764/

[7] H. W. Dettmer, *Goldratt's theory of constraints: a systems approach to continuous improvement*. ASQ Quality Press, 1997.

[8] C. Cetina, "Achieving autonomic computing through the use of variability models at run-time," Ph.D. dissertation, Universidad Politécnica de Valencia, 2010.

[9] J. Muñoz, "Model driven development of pervasive systems. building a software factory," Ph.D. dissertation, Universidad Politécnica de Valencia, 2008.

[10] D. Marples and P. Kriens, "The open services gateway initiative: an introductory overview," *IEEE Communications Magazine*, vol. 39, no. 12, pp. 110–114, Dec 2001.

[11] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks. POLICY '04*, June 2004, pp. 3–12.

[12] J. J. Alferes, F. Banti, and A. Brogi, *Logics in Artificial Intelligence: 10th European Conference, JELIA 2006 Liverpool, UK, September 13-15, 2006 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ch. An Event-Condition-Action Logic Programming Language, pp. 29–42.

[13] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Software & Systems Modeling*, vol. 15, no. 1, pp. 31–69, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0394-9

[14] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language 4th Edition*. Addison-wesley, 2016.

[15] "The eclipse project: Emf model query." [Online]. Available: https://projects.eclipse.org/projects/modeling.emf.query

[16] V. R. Basili, "The role of experimentation in software engineering: Past, current, and future," in *Proceedings of the 18th International Conference on Software Engineering*, ser. ICSE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 442–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=227726.227818

[17] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*. Wiley, 1994.

[18] M. O. B. G. H. Travassos, "Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering," in *In Proceedings of the Workshop on Empirical Studies in Software Engineering (ESEIW)*. IEEE Computer Society, 2003.

[19] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, no. 2, pp. 131–164, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9102-8

[20] M. Ehrig, A. Koschmider, and A. Oberweis, "Measuring similarity between semantic business process models," in *Proceedings of the Fourth Asia-Pacific Conference on Comceptual Modelling - Volume 67*, ser. APCCM '07. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2007, pp. 71–80. [Online]. Available: http://dl.acm.org/citation.cfm?id=1274453.1274465

[21] P. Graf and K. D. Muller-Glaser, "Dynamic mapping of runtime information models for debugging embedded software," in *Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*, June 2006, pp. 3–9.

[22] S. Maoz, "Using model-based traces as runtime models," *Computer*, vol. 42, no. 10, pp. 28–36, Oct 2009.

[23] L. Bodenstaff, A. Wombacher, M. Reichert, and R. Wieringa, "Made4ic: an abstract method for managing model dependencies in inter-organizational cooperations," *Service Oriented Computing and Applications*, vol. 4, no. 3, pp. 203–228, 2010. [Online]. Available: http://dx.doi.org/10.1007/s11761-010-0062-7

[24] O. Nierstrasz, M. Denker, and L. Renggli, *Model-Centric, Context-Aware Software Adaptation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 128–145. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02161-9_7

[25] L. Hamann, M. Gogolla, and M. Kuhlmann, "Ocl-based runtime monitoring of jvm hosted applications," *Electronic Communications of the EASST*, vol. 44, 2011.