Report

for the project seminar:
Online Monitoring of Complex Conditions for Event-based Distributed Architectures

Leonardo Hübscher
Maximilian Völker
Hasso Plattner Institute, University of Potsdam, Germany

Summer semester 2019

1 Introduction

With the ongoing digitization of all areas of our life the amount of developed software systems increases. The size of the hardware is becoming smaller, while the capabilities grow. This allows the use of very small components, e.g. in the form of sensors, in distributed software systems. Such systems are used in the Internet of Things, smart factories, cars, and nowadays also in smart medical environments.

This work is mainly motivated by the overall security of such systems. For us this means mainly two things: data security and safety. Lets take the car with all its sensors as an example: Firstly, nobody should be able to reconstruct where a person was going in terms of data security and privacy. Secondly, all systems should work reliable and in a robust way to prevent malfunctions and accidents in terms of overall security. Especially in the domain of aviation, medicine and power supply the software has to be very robust and secure. Basically in all areas where the programs need to work correctly, securely, and reliably we have to make sure that they are functioning as expected.

However, because of the increasing complexity of distributed software architectures, it is very difficult to assess the correctness of a program. Additionally, there is a trend of dynamically growing distributed systems that makes controlling them even harder. To maintain a high security standard in the areas of risk, the official authorities typically require that the highly security relevant software parts are checked and verified. In this work we will focus on online monitoring (OM) that could be an approach to tackle this challenge.

In the following section we provide more detailed insights into OM and briefly introduce two software solutions for OM. Chapter three is about two use cases. We apply OM for each of the use cases to test certain conditions and to evaluate the performance of the tools used. At the end we will summarize our impressions of the tools and OM.

2 Approach

In this section we will explain Online Monitoring in more detail and classify OM in the field of verification techniques. We then introduce two high level specification languages QTL and MFOTL that are used for defining properties. We will then present two related tools, DejaVu and MonPoly, that implement these languages.

2.1 Online Monitoring

Online monitoring is a lightweight verification technique in the sense of property formulation and model knowledge. Since it is independent of code changes, it even works when the source code grows, which is one of the main advantages. OM determines whether a single run of a software system violates or satisfies a certain set of properties. The properties that get verified using run-time verification are specified using a high level specification language, that expresses the expected occurrence or absence of certain events. Software components can emit such events, indicating that they have started or completed a certain action. A text processing software could for example emit events if a file was opened, updated or deleted. Often, events are stored in log files for later analysis. Based on the logs one has to define the signatures of the events first. This can be either done by looking on the log files or by gathering knowledge about the software system, e.g. by reverse engineering. The last preparation step includes the formulation of the properties that are based on the events as input. Checking an event stream happens either on recorded executions or in real-time. The former gets called *offline monitoring*, while the latter is about *online monitoring*, where the traces of executions have to be processed in an incremental and efficient way.[9]

The *monitor* is a mandatory component of the run-time verification. A monitor is defined as "[...] a device that reads a finite trace and yields a certain verdict"[5]. It runs in parallel to the inspected software and is used to check if an execution of a program, that has a certain trace of events, satisfies a given property formulation. The monitor needs to work fast and use very few resources in order to not influence the running system. Since the trace is time-dependent, the temporal logic that is being used to describe the properties requires the monitor to be impartial and anticipated. This basically means, that the monitor guarantees that verdicts are not answered too fast (before the system can be sure) or too late (when the system could have known before for sure). However, this also requires the verdict to be capable of being more than just *true* and *false*. Since there are situations, where the system just can not know the answer yet, at least a third state, called *inconclusive* is necessary. In the literature it is also stressed that the monitor is just for detecting such violations and not reacting on them. However, the results of the monitor can be used as a base for further processing. [9]

2.2 An Overview of the Verification Techniques

The most formal verification technique is *theorem proving*, which is mostly done semi-automatically. Theorem proving is used to proof the correctness of a system and its granularity is comparable to a mathematical proof. A person who has to verify a software typically uses tools that are producing proofs and checks these proofs for correctness. A less formal way is *model checking*. In contrast to theorem proving it can be done fully automatically and is applicable when it comes to very small and simple systems, where the software inspector can fully understand the whole software behavior. The least formal way of reviewing a software systems behavior is *testing*, e.g. by using unit tests and thus working on code-level. It is an incomplete approach to show the correctness of a specific part of a software. Each test checks whether the system returns the expected output for a given input. Therefore, the input domain and the behaviour are often only partly covered.[9]
Online monitoring, also known as *run-time verification* tries to combine model checking and testing. We want to compare OM with model checking and testing briefly.

2.2.1 Online Monitoring Compared to Model Checking

OM is less formal than model checking. While model checking is about checking all executions, OM checks only the ones, that have been monitored in an actual run of the software. As soon as the used programming language supports the concept of loops, model checking has to cope with potentially infinite traces. OM is applied on systems that are supposed to keep running. However, it only works with the actually observed traces, which are finite by definition. On the one hand, model checking requires knowledge about the software module that is being checked. On the other hand, OM does only require very little, up to no knowledge about the model, and is therefore applicable on external library code where the source is unknown (also called *black box library*). Lastly, OM only scopes on specific variables that are of interest. In model checking that is not possible, hence the whole state of each step has to be taken into account. Having to keep track of all variables while having loops can also lead to the *state explosion problem*, since it is required to keep a snapshot for all variables for each iteration.

2.2.2 Online Monitoring Compared to Testing

Online monitoring and testing do not test all possible paths of the model. Testing is not used to verify the correctness of a whole property, it is more about point wise checks. However, sometimes the source code of the software that is being tested needs to get changed to make testing possible. This is a huge drawback compared to OM, since there no code changes are required. OM can be considered as *passive testing* and is not as granular as testing, as it works with complete properties. In contrast to OM, testing can be used to verify the correctness of edge-cases which are very unlikely to happen during run-time.[9]

2.2.3 Online Monitoring Classification

Online Monitoring should be used, when someone has to test a black box library, or if information becomes available at run-time only, such as in dynamically changing event-based distributed architectures. In these architectures, the behaviour of the application depends heavily on the environment. Self-managing systems that are adapting their behaviour depending on the environment, which might change over time, are hard to test with the common approaches. Online Monitoring can also be used in security-critical systems as a second verification.[9]

We can conclude that "Online monitoring is a powerful technique that tries to accomplish an optimal trade-off between testing (incomplete) and more formal methods (often unfeasible)"[6]. However, only the monitored errors are observed. Online Monitoring used exclusively, does not provide evidence for the correctness of a whole system.[9]

2.3 Temporal Logic

To be able to check the properties, the monitor component (see section 2.1) needs them to be in a formalized and thus machine-understandable way.

With *first-order* logic we can express properties, that are either *true* or *false*, in a formal and mathematically defined way. [4] For example, we can express that only the user *admin* can modify files:

$$\forall\, f,u : modify(f,u) \implies u = "admin"$$

Where f (file) and u (user) are variables of a certain domain and *modify* is an event, emitted by the monitored system. Using such first-order logic expressions, the state of a whole system can be described.

Temporal logic extends this by adding operators which not only allow statements about the current state, but also about changes over time. [4] Now we can extend the property and specify that files can only be changed if they were not previously marked as read-only:

$$\forall\, f,u : modify(f,u) \implies u = "admin" \wedge \neg\, \blacklozenge read\_only(f)$$

Operator	Symbol	Meaning
Future	$\Diamond$	Eventually it will happen
Past	$\blacklozenge$	It happened at some point before

Table 1: Temporal operators and their symbols

In this work, we use the temporal operators and symbols as shown in table 1. More detailed information about first-order logic or temporal logic, as well as examples and explanations can be found in [4].

2.4 Online Monitoring Tools

There are several software solutions available that are capable of monitoring event streams. For this assignment we have used two programs that are freely available for non-commercial use: *MonPoly* and *DejaVu*. Both are briefly introduced in the following sections, including their characteristics and differences.

2.4.1 MonPoly

MonPoly[1], available under the GNU LGP License 2.1, was developed at the ETH Zurich and is programmed in OCaml. It accepts formulas using *metric first-order temporal logic*: Temporal operators, as introduced in section 2.3, can be further specified using time intervals. E.g., it is possible to express that something must have happened at least last two hours before or must not happen within the next 30 seconds. For example, we could express that the read-only marking in the property introduced in section 2.3 only lasts for two hours:

$$\forall\, f, u : modify(f, u) \implies u = \text{"}admin\text{"} \wedge \neg \blacklozenge[0, 2h] read\_only(f)$$

Now, a file can be modified by an admin, as long as there was no *read_only* event for this file in the last two hours.

More general information about MonPoly can be found in [3]. The algorithm of the software is described in [1].

2.4.2 DejaVu

The other tool used in this paper was developed at the California Institute of Technology and is written in Scala. In contrast to MonPoly, DejaVu[2] only works with formulas that are expressed in *first-order past time linear temporal logic*. Consequently, DejaVu only accepts past time temporal operators, e.g. $\blacklozenge$, but not $\lozenge$ and it does not support the *metric* aspect like MonPoly. Therefore, the example with the two-hour marking could not be expressed either. Detailed information on DejaVu can be found in [7].

[1] `https://sourceforge.net/projects/monpoly/`
[2] `https://github.com/havelund/dejavu`

3 Use Cases

The overall goal of the project seminar was to gain a general understanding of temporal logic, online monitoring, and the tools MonPoly and DejaVu. In order to direct the work and limit the scope, an assignment was posed. This assignment included the specification of two data sets, as well as respective conditions expressed in natural language, which should be checked with the tools. The complete assignment is included in the appendix.

In this section we first present the setup used for evaluation and our general workflow we applied to each data set. Then, we will investigate the two use cases, where we want to apply online monitoring (OM) on. The first use case deals with the topic data security / privacy and is based on a real-world data collection campaign, ran by Nokia. The second use-case is about a synthetic data set around the security-critical administration of antibiotics in a smart medical environment. For each use case we will analyze the given data set according to the event types, to get a better understanding of the log files. We will assume that an entity-relationship model can be applied to the logs and used for our online monitoring approach. Furthermore, we will use the previously mentioned tools, DejaVu and MonPoly, to test several given properties. In the end we will conclude this section by evaluating the performance of each tool. All scripts, data analyses and logs have been uploaded to GitHub[3]. The logs of the first use case can be downloaded separately[4].

3.1 Evaluation Setup

In order to ensure a mostly uniform and unimpaired execution of the tools throughout all test runs, we decided to use a virtual machine as basis.[5] Therefore, all executions were run on the same underlying hardware, without any highly demanding, parallel tasks. To measure run times and memory consumption, the tool *pidstat*[6] was used, set to a sampling interval of one second. We decided to use an external tool instead of any internal measurements, like MonPoly offers, to have a consistent, comparable, and external view of the resource requirements of each tool. Also, it allows us to trace the memory consumption over time. With memory consumption we refer to the *virtual memory size*, which we used since it also includes memory in the swap file. As measuring the "real" memory consumption of a process is quite difficult, the numbers provided are rather interesting for comparisons, but should not be considered in isolation. Even though each task was executed separately, there are always background processes with resource consumption. Therefore, the measured execution times of the tools are only samples and repeating the same check on the same machine can lead to slightly different numbers.

[3] `https://github.com/deeps96/Online-Monitoring-of-Complex-Conditions-SS-19`
[4] `https://www.idiap.ch/dataset/mdc/download`
[5] DigitalOcean Droplet with 1 CPU, 2 GB of memory (+ 3 GB swap) and 50 GB SSD
[6] `http://sebastien.godard.pagesperso-orange.fr/man_pidstat.html`

3.2 From Analysis to Evaluation

The following paragraphs describe the steps we have carried out for both of the two use cases.

Analysis The first step is it to get to know the data set in more detail. We used a Jupyter notebook (Python) in order to learn about events included in the data set, their structure and occurring attribute values. In addition, various visualizations of values and their distribution within the data are created by the notebook. After the extraction of the event types, the meaning of the individual attributes had to be defined manually and with this step, the relationships between the entities were also clarified.

Formalization After the definition of the event types and their attributes, the properties given in natural language had to be formalized using temporal logic (see section 2.3). Since both MonPoly and DejaVu have their own tool-specific syntax, the formalized properties had to be transposed into the respective "language". Due to the complexity of some formulas, we often started with basic assertions instead of the whole property and checked whether this part is executable or not and added more and more checks step by step. To ensure that we receive only correct violations at each step, we have developed test data sets for each use case. These included some violating events, as well as valid events, to see what cases the property already covers or whether it reports events falsely. This step was necessary because running the formulas for each development step on the entire data set would have taken too long.

Execution Before the tools could be executed, the data sets had to be transformed to the event log formats the single tools need. We therefore wrote several scripts to "translate" the given data, including a script to transform a MonPoly-style log to the DejaVu format. Then, the tools were executed on the tool-specific log file using the formulas created in the previous step while measuring their performance with *pidstat* (as described in section 3.1).

Evaluation Both tools create files with a list of events that violate the respective property. To compare the measurements, we wrote a script to re-format the output of *pidstat* to the machine-readable csv format. We then used Excel and Tableau Desktop[7] for evaluation and plotting.

[7] `https://www.tableau.com/products/desktop`

3.3 Use Case: Lausanne Data Collection Campaign (LDCC)

The Lausanne Data Collection Campaign was issued by Nokia and ran from 2009 to 2011. During the campaign, data was collected using the sensors of smartphones. The data collection got triggered by the rise of data driven services that are used to anticipate the behavior of individuals in social networks. However, the campaign was not just about collecting the data, but also testing how the user can be empowered to manage his own data in terms of data privacy. Each participant could delete his/her own data. About 180 people participated in the event, which resulted in a data set size of 28.2 GB. The data set is available for non-profit organizations.[8]

3.3.1 A Brief Analysis of the LDCC Data Set

With a size of nearly 30 GB analyzing the data in-memory is not feasible without the proper hardware. This is the reason why we decided to use Jupyter notebooks in conjunction with a MongoDB instance, to analyze the data set. With that setup we were able to take a closer look, on how the data looks like, which is a necessary step in order to define the event types and properties. The LDCC data set consists of 218,714,631 events, distributed over nine different event types. The events can be separated into *database operation events (blue)* and *script events (orange)*, as shown in figure 1. All of them are representing several entities, no relations are expressed between them by the events.

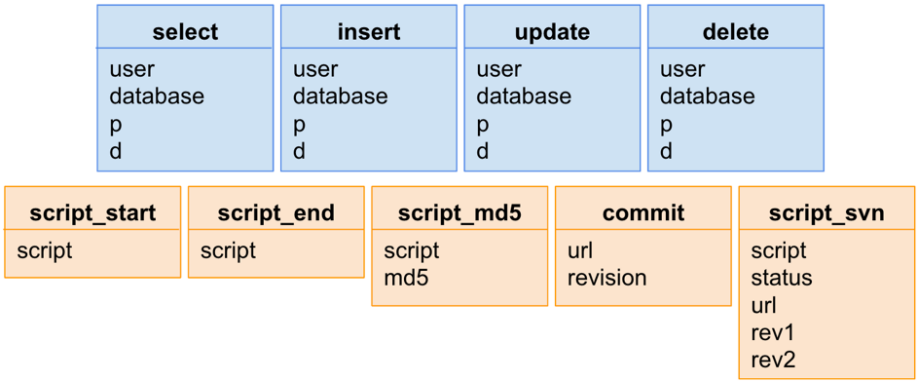

Figure 1: Event types and parameters as entity-relationship model for the LDCC use-case.

A closer look on the event parameters provides first insights, that might proof useful for checking our following results. As shown in table 2, most of the attributes, like *user*, *database*, *script*, or *status* have a very small value range. On the other side, the parameters *d*, *url* and *revision* seem to reflect some sort of unique identifier. We also learn, that we have to catch the value *[unknown]* when doing an equal-match.

Lastly, we can verify that there are only three databases we are working on: db1, db2, and db3.

Database Operation Events			Script Events		
	count	**example**		**count**	**example**
user	62	script, triggers, user1...user58, [unkown]	**script**	3	script1, script11, script12
db	3	db1, db2, db3	**md5**	5	longer strings
p	215	various numbers, [unkown]	**status**	3	svn-latest, unkown
d	+++	various numbers, [unkown]	**url**	+++	arbitrary strings
			rev1	4	251, -1, 224, 394
			rev2	3	-1, 205, 224
			revision	+++	arbitrary numbers

Table 2: A closer look at the parameters of the LDCC events.

Figure 2 shows the distribution of event types across the whole data set. As you can see, the *insert* event type is clearly dominating the log with a share of 98%. This is comprehensible, since we expect a lot of automated insert operations by the data collection software in comparison to the manual delete operations or the automated backup.

3.3.2 The Properties 'script2', 'insert' and 'delete'

Based on the given assignment, we have investigated three properties. The first one checks whether the privileges have been implemented correctly. The second one is about verifying the proper functioning of the backup. The last one is about checking the synchronization between db3 and db2.

1. *script2*
   Only the script script2 may delete data in db2.
2. *insert*
   Data uploaded by the phone into db1 must be inserted into db2 within 30 hours after the upload, unless it has been deleted from db1 in the meantime.
3. *delete*
   Data may be deleted from db3 iff[8] it has been deleted from db2 within the last minute.

[8] if and only if

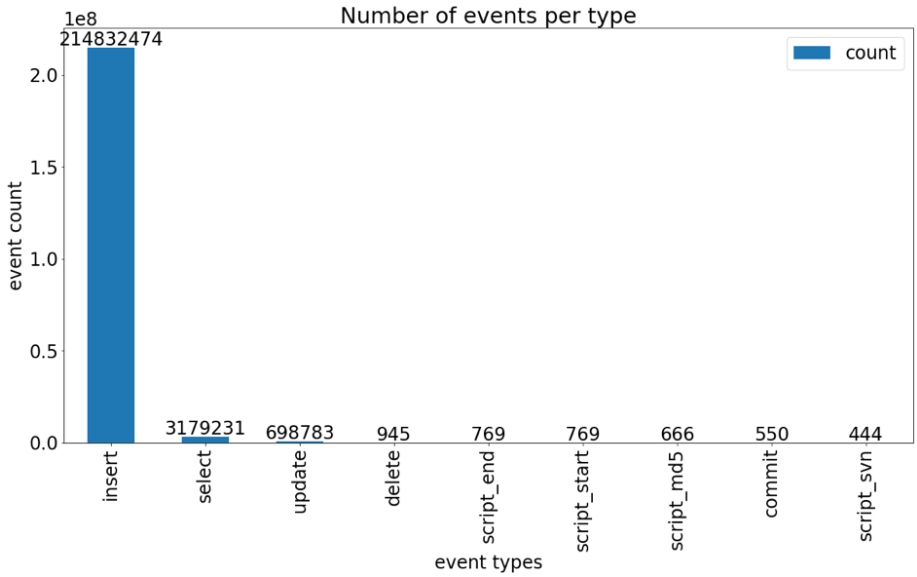

Figure 2: Event type distribution for the LDCC data set

Below are the associated property formulations for *script2*, *insert* and *delete*[9]. As you can see, the first property *script2* is an example for a state property and does not rely on temporal features. This has to be considered, when evaluating the run-time of the tools later.

$$script2 : \forall user.delete(user, "db2", p, d) \implies user = "script2"$$

$$\begin{aligned} insert :& \exists user.insert(user, "db1", p, d) \wedge \neg user = "triggers" \wedge \neg user = "script" \implies \\ & (\\ & \exists user2, p2, d2.\Diamond[0, 30h] insert(user2, "db2", p2, d2) \wedge d = d2 \\ & \bigoplus \\ & \exists user2, p2, d2.\Diamond[0, 30h] delete(user2, "db2", p2, d2) \wedge d = d2 \\ &) \end{aligned}$$

$$\begin{aligned} delete :& \exists user, p.delete(user, "db3", p, d) \Leftrightarrow \\ & \exists user2, p2. \blacklozenge [0, 60s] delete(user2, "db2", p2, d) \end{aligned}$$

When translating the properties to the tool-specific formulas, we made some assumptions in order to simplify the properties:

[9] The checks against the value *unknown* have been removed for clarity.

	script2	*insert*	*delete*
MonPoly	62	0	2 (892 incl. *[unknown]*)
DejaVu	62	/	/

Table 3: Amount of violations reported by the tools for each property

- For *insert*: We assume that if the data was deleted in db1, it can not be inserted to db2 afterwards. Therefore, we can relax the *XOR* from the formula to an *OR*. Now, it would still be possible that the data gets deleted from db1 after inserting it into db2. But we assume that *in the meantime* refers to the time-span between inserting into db1 and into db2. The resulting formula can be found in the appendix, Property 1.
- Concerning *delete*: After consultation with the tutor we relaxed the property to assure that data can only be deleted from db3 if it was deleted from db2 in the last minute too. Therefore, we waived the check that each element deleted from db2 must be deleted from db3 within the next minute. See Property 2 in the appendix for the formula.

3.3.3 Evaluation

Violations In table 3, the number of violations of the three properties reported by each tool is displayed. For the first property, *script2*, MonPoly, as well as DejaVu reported the same number of events violating the property. No violations of the *insert* property were found by MonPoly, and the tool reported only two contravening events for the third property, *delete*. As described in table 2, there are also *[unknown]* values for attribute d, which links entries between the different databases. If the value *[unknown]* is included in the checks, MonPoly reports 892 violations for *delete*. But as we can not be sure, that an entry with $d = $ *[unknown]* in db2 really links to all entries with $d = $ *[unknown]* in db3, we excluded this value for the subsequent analysis. As shown in table 3, the last two properties could not be evaluated with DejaVu as both properties include time window restrictions, which are not supported by the logic DejaVu uses.

Tool Comparison Figure 3 shows the memory usage and execution time of MonPoly and DejaVu for the property *script2*, the only property both tools were able to evaluate. MonPoly requires approximately 3 MB more memory than DejaVu, but was able to finish its analysis in about 200 seconds (26 percent) less time. Also, both tools have a constant memory consumption, this might be related to the nature of the property *script2*: As a state property (it has no temporal operators), the programs can check each event in isolation and do not have to remember former states.

The complete list of measured run times and memory consumption for MonPoly (on

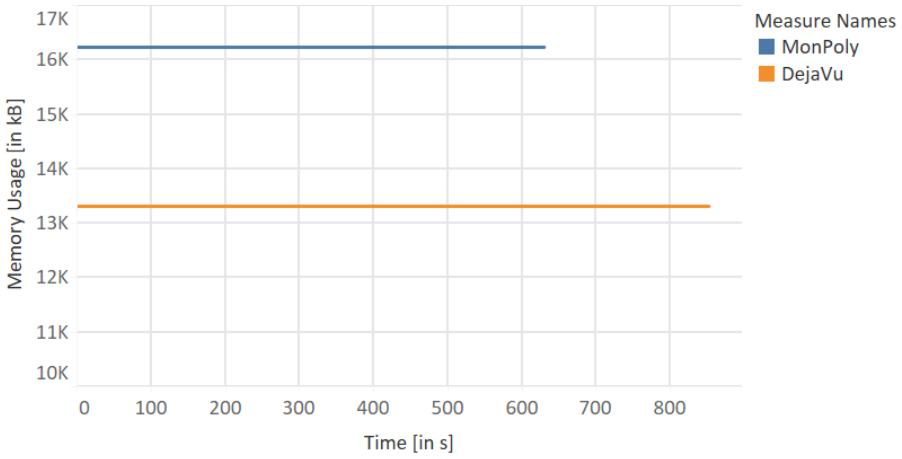

Figure 3: Run time and memory consumption on property *script2*

all three properties), as well as for DejaVu (on *script2* only), can be found in the appendix, table 5.

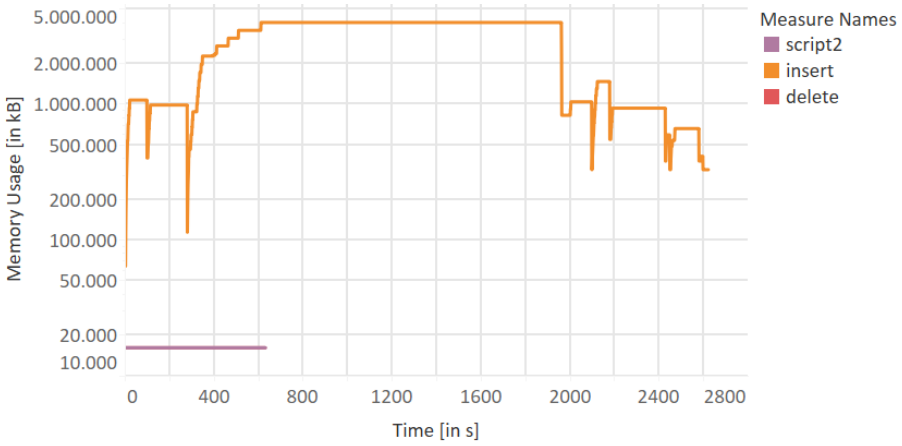

Figure 4: Run time and memory consumption of MonPoly on all three properties (*delete* ≈ *script2*). Logarithmic scale on memory-axis.

MonPoly Comparison Figure 4 compares the MonPoly executions on all three properties. There are two noticeable findings: First, the executions on the first and third property are very similar, both in terms of memory usage and run time. At a fleeting glance, this might be surprising, because, as mentioned before, *script2* is a simple state property, whereas *delete* includes a temporal operator. But considering

that the data set was collected over a large period of about two years, a one minute time window (as it is defined in the property) might not contain so many events that it would noticeably affect the memory usage. Secondly, the execution of the second property, *insert*, shows a significantly different behavior with regard to time and memory requirements. Checking the data set takes much longer and, in the peak, about 250 times more memory.[10] In addition, the memory consumption is not constant over time, but fluctuates widely. Most likely this is due to the high complexity of the property and the two 30 hours windows in the formula. The "memory plateau" in the middle of the execution could be caused by an uneven distribution of events within the data set. However, investigating the correlation of the temporal course of memory consumption with the distribution of the different event types in the data set is beyond the scope of this work and must be left for future work.

3.4 Use Case: Smart Medical Environment

The scenario of this use-case is about dental operations in a medical environment that uses modern technology like smart pumps and several sensors. Dental operations are increasing the risk of infective endocarditis, which is an infection of the inner surface of the heart of a patient.[10] The disease is caused by a bacterial infection. A dental operation abets the occurrence of the disease, since it is possible that oral flora enters the bloodstream during a dental operation. This can be a risk, especially for elderly. However, preventive antibiotics are given to lower the likeliness of infective endocarditis. In this scenario, the antibiotics are injected using smart pumps.

3.4.1 A Brief Analysis of the Synthetic Data Set

While we had to deal with quite big logs in the LDCC use-case (section 3.3.1), the synthetic data set is much smaller. It consists of nine files, each file with about 7,300 up to 90,000 events resulting in an overall event count of 365,940. The event types allowed us to derive an entity-relationship-model that provides a better overview. The model is shown in figure 5.

Each of the nine event types belongs to either an action, a dental operation, a patient or a smart pump or a relation between them. Of greater importance for the properties will be the relation-events. They link two entities with each other. Interestingly, the parameters are mostly unique numbers. Only the name of the patient and the type of the action have a smaller value range. The name has only three possible values: *Alice*, *Bob* and *Charlie*, and the type can either be *noise* or *antibiotics*.

Figure 6 shows the distribution of the event types in the synthetic data set. It immediately catches the readers eye, that there are much fewer dental operations

[10] Since the script uses more than 2 GB of memory, it must use the swap file. These additional read and write operations can have a significant effect on the run time.

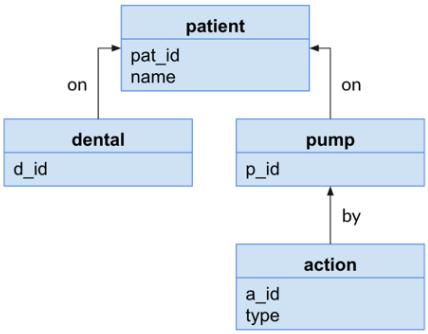

Figure 5: Event types and parameters as entity-relationship model for the medical use-case

performed on patients, than we have patients and actions applied to them. This could be due to noise that got added when creating the data set, since the data is taken not from the real world.

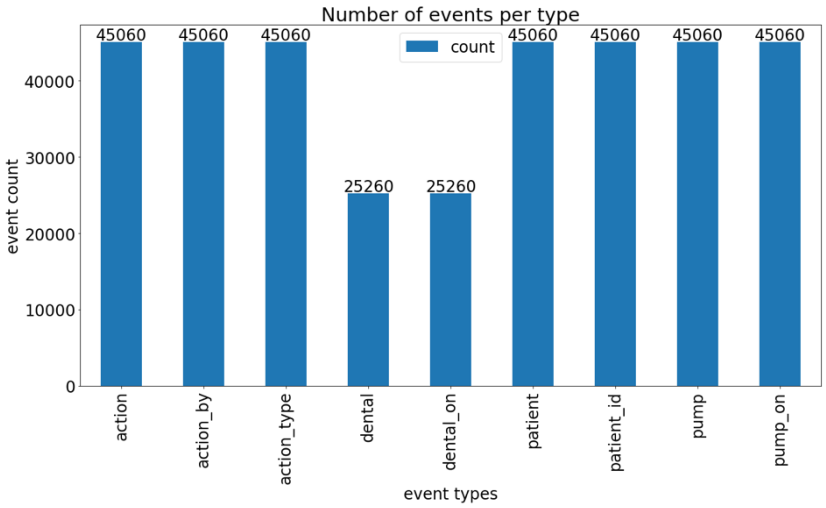

Figure 6: Event type distribution for the medical data set

3.4.2 The Properties 'before' and '2hrs'

For this use case, we are investigating two properties. The first one ensures, that each patient gets antibiotics before a dental operation. The second one tries to catch the cases, where the antibiotics have not been administered before the treatment. In this case it should also be possible, to give the antibiotics up to two hours later.

However, the patient should not get double the dose.

1. *before*
   If a patient is scheduled for a procedure, an antibiotic for prophylaxis should be administered by a smart pump before the procedure.
2. *2hrs*
   If the antibiotic is not administered before the procedure, it may be administered up to 2 hours after the procedure. However, administration of the antibiotic after the procedure should be considered only when the patient did not receive the pre-procedure dose.

As shown in section 3.4.1 the logs do contain entities and relationships. While formulating the properties, we had to decide whether we check the existence of the entities of a relation. Since we do not have any other properties defined beforehand which validate the entity existence, nor can we assume that all relations are well formed, we added these checks to the properties. The property formulations can be found in the appendix, Property 3, Property 4.

3.4.3 Evaluation

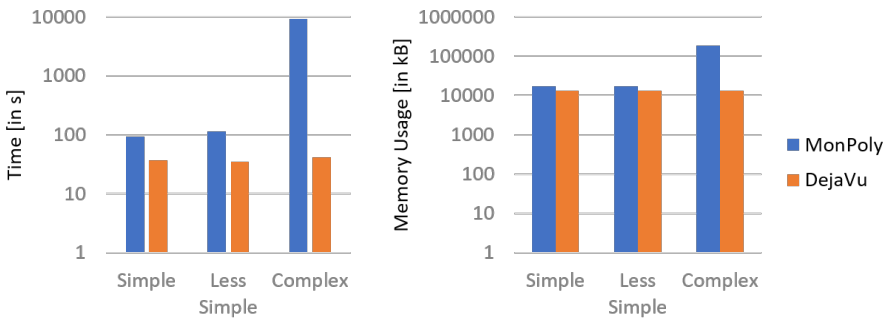

Figure 7: Run time and memory consumption in comparison to complexity of formula. Time and memory axes shown on logarithmic scale.

Formula Complexity In the first run, we tried to evaluate the property *before* with the entities and their relations. But we soon realized, that this has a great impact on the run time, especially for MonPoly. Therefore, we experimented with three formulas with different degrees of assumption:

simple Checks only the relational events: *pump_on, action_by* & *action_type* before *dental_on*.

lessSimple Like **simple**, but includes the check if *dental* occurred before *dental_on*.

	1	2	3	4	5	6	7	8	9
MonPoly	5029	66	5029	531	60	532	1000	537	533
DejaVu	5029	66	5029	531	60	532	1000	537	533

(a) Property *before*

	1	2	3	4	5	6	7	8	9
MonPoly	$\perp$	63	$\perp$	452	52	465	1000	474	458
DejaVu	/	/	/	/	/	/	/	/	/

(b) Property *2hrs*

Table 4: Amount of violations reported by the tools for each property.

complex The "original" version, includes all checks: *pump, action* & *dental* must occur.

We run all three versions on the shorter log files 5 to 9. The mean of run time and memory usage over the respective executions is shown in figure 7. For DejaVu, the increasing complexity of the formula seems to have almost no effect on run time or memory consumption. But with MonPoly, even if *simple* and *lessSimple* require almost the same amount of time and memory too, the gap to the complex formula is much larger: MonPoly requires about ten times more memory and even 100 times more time to check the complete property, in opposite to the "relations-only" formula. Due to this fact we decided to run the following evaluations on the simple property for MonPoly and we therefore assume that all required entities exist. To be on the safe side, additional properties could be checked to ensure that when a relational event occurs, the linked events have occurred previously. For example, a new property could be formulated, that verifies that if *dental_on* occurs, the referenced *dental* and *patient* events have happened before. However, for DejaVu we have retained the complex version since it seems to run well and allows us to compare the performance with respect to found violations by the simple and complex property.

Violations In table 4, the number of violations found by MonPoly and DejaVu on each property and log are shown. For the first property, *before*, both tools coincide in the amount of events that infringe the property. The fact that some numbers are the same between the different log files may be due to the fact, that we are working with a synthetic data set. For the second property, shown in table 4b, again only MonPoly was able to analyze it, as it includes time conditions which DejaVu can not handle. Also, we can not report any number of violations for logs one and three, due to run times of MonPoly with more than 80 hours. We have stopped the analysis here, since run times of this size can be considered impractical for real-world use cases.

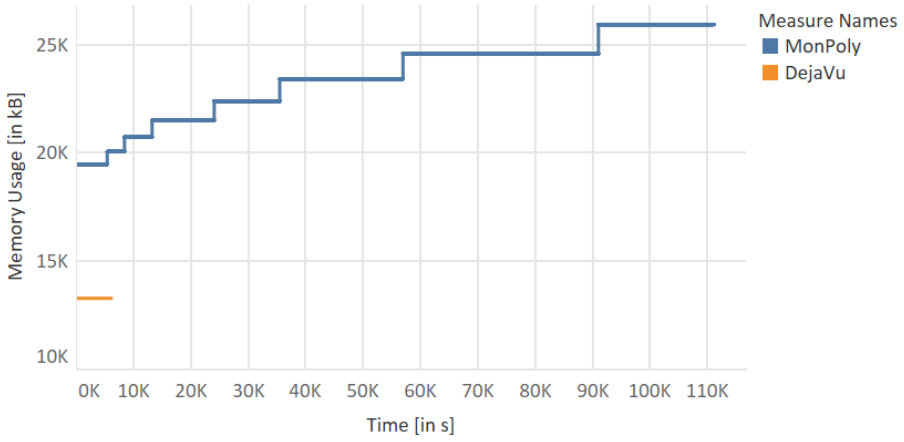

Figure 8: Run time and memory consumption of MonPoly and DejaVu on property *before* on log 1

Tool Comparison Again, we compare the run time and memory consumption of both tools for the first property, *before*. The execution on log 1 is considered in figure 8, as it is one of the largest logs in this data set. In opposite to the first use case where both tools behaved quite similar (see section 3.3.3 and figure 3), there are now clear differences: The memory usage of MonPoly is no longer constant, but increases over time. At its peak, it requires almost twice as much memory as DejaVu. Additionally, DejaVu took about 1.69 hours to analyze the 90090 events in the log, whereas MonPoly took more than one day to execute (30.83 h). Considering that MonPoly even only checks the simplified version of the property, this difference is even more drastic.

The complete list of the measured run times and memory consumption of MonPoly and DejaVu for the property *before* can be found in the appendix, table 6. The measurement results for the second property *2hrs* (MonPoly only) which are not further evaluated here, are also included in the appendix (see table 7).

Impact of number of events In figure 9, the run times of MonPoly and DejaVu on the first property, *before*, on all nine logs are shown, along with the number of events in each log. In all cases, MonPoly requires more time to check the property than DejaVu. A conspicuous observation is the strong increase in run times from log 5 to 4: The number of events increases by a factor of ten (approx., from 7290 to 72090), but the run time of DejaVu rises from 34 seconds to 3072 seconds (factor of $\sim$100) and MonPoly now even requires 51645 seconds, instead of 50 seconds (factor of $\sim$1000). This might be caused by the fact, that the property requires at least three (in the simplified version) different events to happen, before *dental_on* occurs. Therefore, the tools have to handle a much larger "event-space", than in the smaller logs, in order to check if all preconditions for all *dental_on*s were fulfilled.

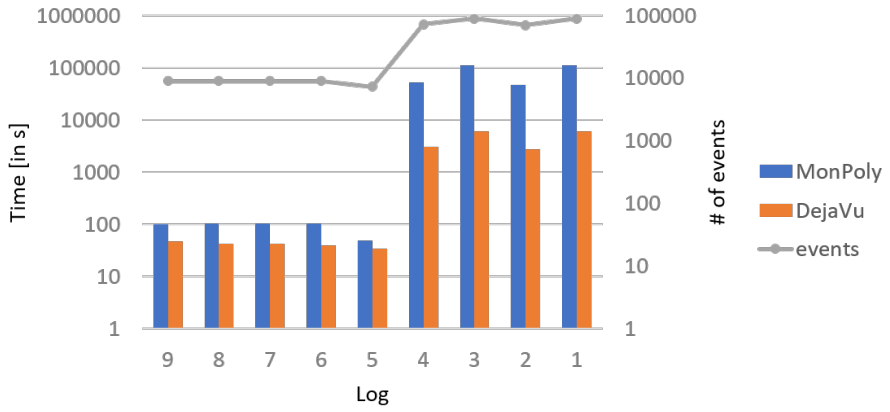

Figure 9: Run time and memory consumption in connection with the amount of events. Both y-axes are represented on a logarithmic scale.

Optimizing DejaVu DejaVu offers a parameter *bitsPerVariable*, which determines how many bits are reserved per variable. A value of 20 (which is the default value) means, that for each variable in the formula, 2^{20} different values can be stored. As the largest log files in this use case contain at most 10010 different values for each event type (e.g. *patient_id* or *action_id*), there might potential for optimization, as 14 *bitsPerVariable* are enough to allow for 16384 different values[11].

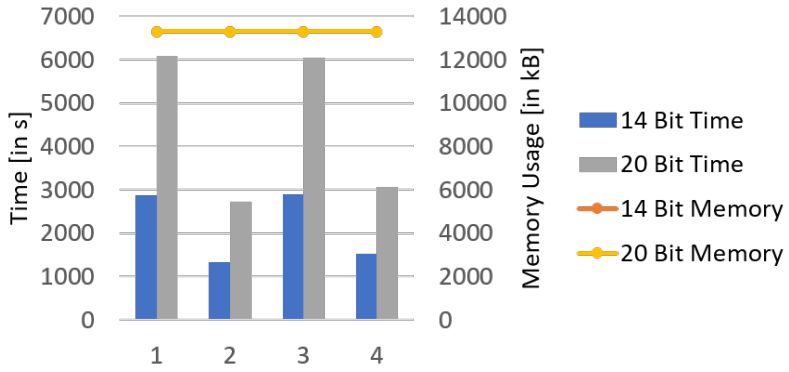

Figure 10: Run time and memory consumption of DejaVu on log 1-4, property *before* with *bitsPerVariable* set to 14 and 20 (default)

Figure 10 shows the results on log 1 to 4 with the *bitsPerVariable* parameter set to 14 and the default value 20. An interesting finding is that the memory consumption of DejaVu is apparently not influenced by the number of bits per variable, which

[11] $2^{13} = 8192 < 10010$

would be the first intuitive thought. Most likely DejaVu reserves a certain amount of memory for execution, how much it actually uses cannot be determined. However, the execution time is heavily influenced by the bit: The run time was halved for each execution on the first four logs. This is consistent with the statement in DejaVu's read-me: "A too high number can have impact on the efficiency of the algorithm."[12] Nevertheless, it was surprising that the run time depends so much on this parameter.

We also analyzed logs 5 to 9 with bit 14 and added an analysis with bit 10 for these five smaller logs, as they contain at most 1010 different events of each type, where $2^{10} = 1024$ is sufficient.[13] In this case, the run times where reduced by only about one third. However, since these executions were in general quite short (less than 1 minute), the initialization time required by the tool, which is independent of the bit parameter, must also be taken into account. The difference between *bitsPerVariable* of 14 and 10 is, at least in this case, negligible. In the appendix, figure 13 shows the graph for these executions, similar to figure 10. The numbers on run times and memory consumption for these evaluations, including log 1 to 4 can be found in table 8 (appendix).

We decided to evaluate the parameter mainly with this use case, as we only have a state property for the first use case, where DejaVu has not to keep track of former values. But for completeness, we also evaluated the first use case using a wide range of different bit settings, the results can be found in the appendix (see figure 14). There, the run time also increases slightly with the *bitsPerVariable*, the memory consumption remains the same.

[12] `https://github.com/havelund/dejavu#running-dejavu`
[13] As expected, executing DejaVu with 10 bits on log 4 led to an "out of memory" error.

4 Discussion

In this section a summary of our evaluations and performance measurements are presented, as well as a few thoughts on the tools.

4.1 Overall Performance Comparison

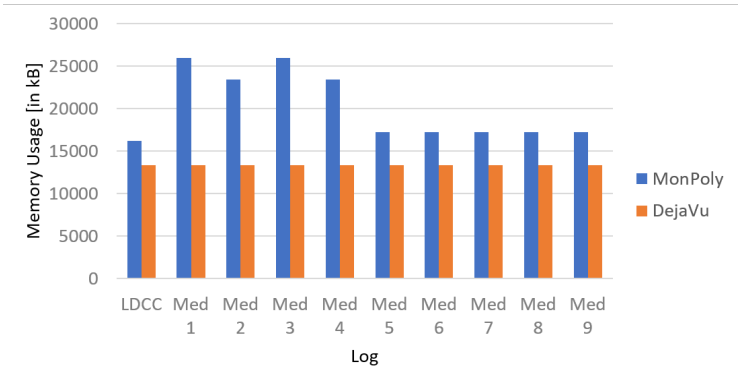

Figure 11: Overall (max.) memory comparison

Memory In figure 11, the maximal memory consumption of MonPoly and DejaVu on all comparable executions (the first property of both use cases) is shown. The first noticeable insight is, that MonPoly requires more memory in all cases, regardless of the use case/data and needs a maximum of twice as much memory as DejaVu. In addition, the amount of memory required by DejaVu is always the same (13312 kB) and always constant during execution. This suggests a better memory management by DejaVu, which uses binary decision diagrams (BDD) and has an internal garbage collector to re-use no longer needed BDDs[14]. MonPoly however relies on an OCaml standard library which uses balanced binary trees [2]. Interestingly, there is also a former MonPoly implementation, based on BDDs, but its development seems to be discontinued. More theoretical thoughts about the memory consumption of MonPoly can be found in [2], too.

Time Figure 12 shows the same evaluations but plots the time needed to execute instead of the memory consumption. Apart from the execution on the LDCC data set (which was the evaluation of the state property *script2*), MonPoly also always requires more time to finish its analysis. At the maximum, the run times differ by a factor of about 20.

[14] `https://github.com/havelund/dejavu#running-dejavu`

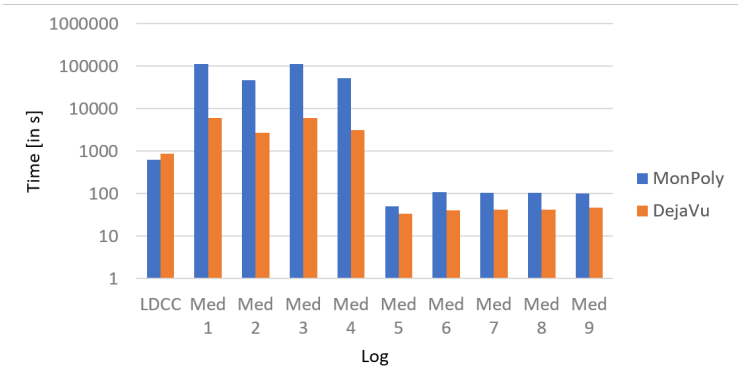

Figure 12: Overall time comparison on first property. Logarithmic scale on time axis.

Summary In our test cases, DejaVu outperformed MonPoly clearly: It requires less memory, even though the difference is not that big, but the main advantage is its much better run time. In the worst case it is not significantly slower, but in (our) best case DejaVu requires 20 times less time, which is a not unimportant factor when large logs have to be analyzed. But, of the five properties from the assignment, only two could be expressed with the logic DejaVu uses. This is a major drawback as any properties with time constraints can not be expressed and verified with this tool, although the time aspect is very important especially in security and compliance related areas: e.g. medication must be administered at certain intervals or the take-off checklist does not have to be completed at some point, but immediately before the start.

4.2 Usability of the Tools

After translating the properties from natural language into formulas expressed in temporal logic, the even more difficult part began: Re-writing them in the tool-specific language. Whereas DejaVu at least has a quite comprehensive read-me, MonPoly lacks any official documentation, apart from some examples and scattered segments in various publications. In order to get to know the keywords MonPoly understands, for example to express *before* ($\Diamond$), we had to look into the source code[15]. In the best case there would be a documentation for MonPoly, which facilitates the initial steps and explains at least the basic operations and parameters like DejaVu has.

Of course, both tools are optimized for run-time and memory consumption and not usability. To speed up the process of writing formulas, both tools offer a *formula checker*. However, they are not very descriptive and are not a great help when it comes to debugging incorrect properties. Especially when it comes to more complex formulas there is the need for better error messages to find incorrectly bound variables or misplaced parentheses.

[15] `https://bitbucket.org/monpoly/monpoly/src/master/src/MFOTL.ml`

5 Conclusion

The report introduced online monitoring (OM) as one possible verification technique that works with distributed event-based architectures. We outlined how the approach differs from other verification techniques in order to give a better understanding about online monitoring. We presented use-cases that reflect two important aspects that can be validated by the use of OM: data security and the overall system security / robustness. While developing solutions for the use-cases, we developed a process, where we first applied some data analysis on the given logs beforehand, which proofed to be very useful regarding the gain of a first understanding at the beginning and the performance evaluation in the later process. Without seeing temporal logic before, it was very interesting to work with the new tool-set to transform the properties into formulas. However, the new syntax did not just introduce more possibilities, but also showed the need to think about bound and free variables in more detail. We recommend a good introduction in this topic, before starting with the actual formulations. After formulating the properties, we transformed them in order to feed them into the tools DejaVu and MonPoly which we used. At first, we struggled a lot because of the lack of documentation, which required us to sometimes look into the source code of the tools, which has been luckily available. During the property translation we missed more expressive error messages, that could have helped with issues regarding the previously mentioned free and bound variables. To overcome this challenge, we mostly used trial-and-error and test runs on our test logs until we found a working solution. This is a perfect example that demonstrates, that these tools are trimmed for performance and not development. On the other hand, it would be nice to have some sort of trace for each violation for debugging purposes. As outlined in the summary of the evaluation, temporal operators must be handled prudently, since they can heavily influence the run-time and memory consumption of the tools. In general, it seems that DejaVu is faster and requires less resources than MonPoly. However, DejaVu does not support the Metric First-Order Temporal Logic. Additionally, MonPoly seems to be more flexible in the sense that it rearranges the given formulas more and is more robust and also offers more functionality. Yet, a final conclusion on the tools remains open, and further research dealing with the concrete code implementation is advised, in order to give a more detailed assessment. In the end, the tools enabled us to successfully apply online monitoring on the given scenarios.

References

[1] BASIN, David ; KLAEDTKE, Felix ; MARINOVIC, Srdjan ; ZĂLINESCU, Eugen: Monitoring of temporal first-order properties with aggregations. In: *Formal methods in system design* 46 (2015), Nr. 3, S. 262–285

[2] BASIN, David ; KLAEDTKE, Felix ; MÜLLER, Samuel ; ZĂLINESCU, Eugen: Monitoring metric first-order temporal properties. In: *Journal of the ACM (JACM)* 62 (2015), Nr. 2, S. 15

[3] BASIN, David ; KLAEDTKE, Felix ; ZALINESCU, Eugen: The MonPoly Monitoring Tool. In: REGER, Giles (Hrsg.) ; HAVELUND, Klaus (Hrsg.): *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools* Bd. 3, EasyChair, 2017 (Kalpa Publications in Computing). – ISSN 2515–1762, 19–28

[4] BEN-ARI, Mordechai: *Mathematical logic for computer science*. Springer Science & Business Media, 2012

[5] FALCONE, Yliès ; NIČKOVIĆ, Dejan ; REGER, Giles ; THOMA, Daniel: Second International Competition on Runtime Verification, 2015, S. 405–422

[6] GIESE, Prof. Dr. H. ; SAKIZLOGLOU, Lucas ; HÄNSEL, Joachim: *Online Monitoring of Complex Conditions for Event-based Distributed Architectures*. University Lecture, 2019

[7] HAVELUND, Klaus ; PELED, Doron ; ULUS, Dogan: First order temporal logic monitoring with BDDs. In: *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design* FMCAD Inc, 2017, S. 116–123

[8] KIUKKONEN, N ; J, Blom ; DOUSSE, Olivier ; GATICA-PEREZ, Daniel ; LAURILA, Juha: Towards rich mobile phone datasets: Lausanne data collection campaign. (2010), 01

[9] LEUCKER, Martin ; SCHALLHART, Christian: A brief account of runtime verification. (2009), S. 293 – 303

[10] WIKIPEDIA: *Infective endocarditis — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Infective%20endocarditis&oldid=907502352`, 2019. – [Online; accessed 24-July-2019]

6 Appendix

$$
\begin{aligned}
insert :& \exists user.insert(user, "db1", p, d) \wedge \neg user = "triggers" \wedge \neg user = "script" \implies \\
& (\\
& \exists user2, p2, d2.\Diamond[0, 30h] insert(user2, "db2", p2, d2) \wedge d = d2 \\
& \vee \\
& \exists user2, p2, d2.\Diamond[0, 30h] delete(user2, "db2", p2, d2) \wedge d = d2 \\
&)
\end{aligned}
$$

Property 1: Property formulation of *insert* when using assumptions

$$
\begin{aligned}
delete :& \exists user, p.delete(user, "db3", p, d) \Rightarrow \\
& \exists user2, p2. \blacklozenge [0, 60s] delete(user2, "db2", p2, d)
\end{aligned}
$$

Property 2: Property formulation of *delete* when using assumptions

	MonPoly		DejaVu	
Property	**Time in s**	**Mem. in MB**	**Time in s**	**Mem. in MB**
script2	630	16.2	852	13.3
insert	2621	4042	/	/
delete	629	16.2	/	/

Table 5: Run time and memory usage of MonPoly and DejaVu on the three properties of the LDCC use case

		MonPoly		DejaVu	
Log #	**events #**	**Time in s**	**Mem. in MB**	**Time in s**	**Mem. in MB**
1	90090	111005	25.9	6088	13.3
2	70290	46662	23.4	2735	13.3
3	90090	109917	25.9	6053	13.3
4	72090	51645	23.4	3072	13.3
5	7290	50	17.3	34	13.3
6	9090	106	17.3	40	13.3
7	9000	103	17.3	42	13.3
8	9000	104	17.3	42	13.3
9	9000	102	17.3	47	13.3

Table 6: Number of events, run time and memory usage of MonPoly and DejaVu on property *before* (medical use case) on all nine logs

		MonPoly	
Log #	**events #**	**Time in s**	**Memory in MB**
1	90090	/	/
2	70290	470	20.2
3	90090	/	/
4	72090	5142	21
5	7290	5	17.2
6	9090	104	17.2
7	9000	100	17.2
8	9000	98	17.2
9	9000	103	17.2

Table 7: Number of events, run time and memory usage of MonPoly on property *2hrs* (medical use case) on all nine logs

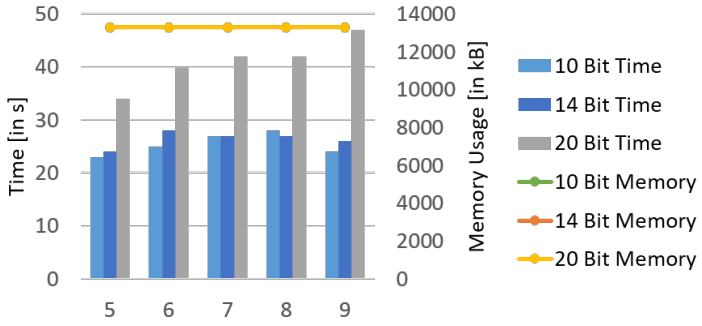

Figure 13: Run time and memory consumption of DejaVu on property *before* (medical use case) on log 5-9, with *bitsPerVariable* set to 10, 14 and 20 (default)

	10 bpV		14 bpV		20 bpV	
Log	**Time**	**Memory**	**Time**	**Memory**	**Time**	**Memory**
1			2870	13312	6088	13312
2			1334	13312	2735	13312
3			2905	13312	6053	13312
4			1532	13312	3072	13312
5	23	13312	24	13312	34	13312
6	25	13312	28	13312	40	13312
7	27	13312	27	13312	42	13312
8	28	13312	27	13312	42	13312
9	24	13312	26	13312	47	13312

Table 8: Run time and memory usage of DejaVu on property *before* (medical use case) on all nine logs with different *bitsPerVariable* (bpV). Time in s & memory usage in kB

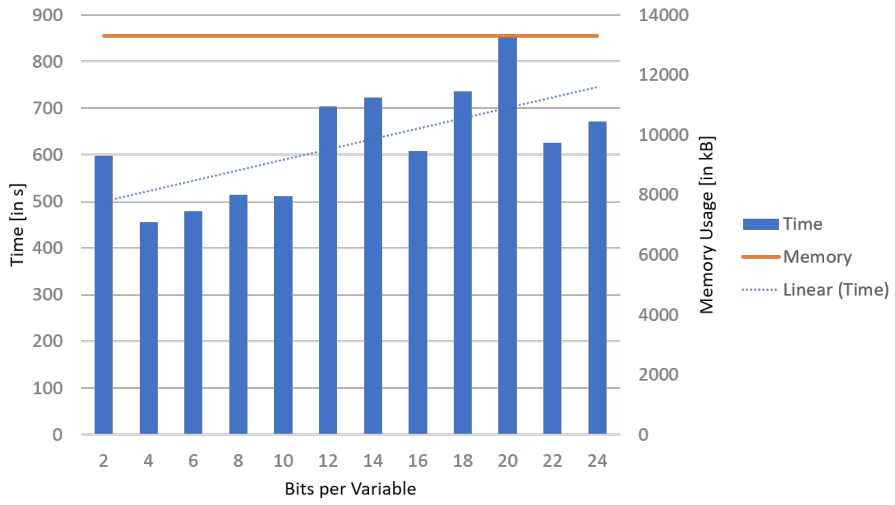

Figure 14: Comparison of run time and memory usage of DejaVu on property *script2* (LDCC) with different *bitsPerVariable* set

$$
\begin{aligned}
before : & \forall pat\_id.\exists d\_id. \\
& dental\_on(d\_id, pat\_id) \wedge \blacklozenge dental(d\_id) \implies \\
& (\exists p\_id, a\_id. \\
& \blacklozenge pump(p\_id) \wedge \\
& \blacklozenge pump\_on(p\_id, pat\_id) \wedge \\
& \blacklozenge action(a\_id) \wedge \\
& \blacklozenge action\_by(a\_id, p\_id) \wedge \\
& \blacklozenge action\_type(a\_id, \text{"}antibiotics\text{"}))
\end{aligned}
$$

Property 3: Property formulation of *before*

$$
\begin{aligned}
2hrs : & \forall pat\_id.\exists d\_id. \\
& dental\_on(d\_id, pat\_id) \wedge \blacklozenge dental(d\_id) \implies \\
& \exists p\_id, a\_id.(\\
& \blacklozenge pump(p\_id) \wedge \blacklozenge action(a\_id) \wedge \\
& \blacklozenge action\_type(a\_id, \text{"}antibiotics\text{"}) \wedge \blacklozenge pump\_on(p\_id, pat\_id) \\
& \bigoplus \\
& \Diamond[0, 2h] pump(p\_id) \wedge \Diamond[0, 2h] action(a\_id) \wedge \\
& \Diamond[0, 2h] action\_type(a\_id, \text{"}antibiotics\text{"}) \wedge \Diamond[0, 2h] pump\_on(p\_id, pat\_id))
\end{aligned}
$$

Property 4: Property formulation of *2hrs*

Assignment

for the project seminar:
Online Monitoring of Complex Conditions for Event-based Distributed Architectures

Lucas Sakizloglou

SoSe '19

Introduction

The goals of the assignment is to i) practice the formulation of properties for online monitoring ii) get familiar with relevant tools and iii) understand factors that impact performance. We will examine two use-cases, that is, two application scenarios where certain conditions are checked during the application execution. Executions will be provided in the form of logs. Checking is to be carried out by two tools: DejaVu [2, 3] MonPoly [1, 5].

Tasks

For each use-case the assignment comprises these tasks:

- Formulate the properties of each use-case in the specification language of each tool (regarding QTL, when applicable)
- Derive the event types
- Check the property against the given log
- Use the tools' output to measure the computation time and consumed memory

Present the results together with your commentary in both the report and the final presentation of the project seminar.

Use-cases

The first use-case scenario and data is based on a real-world data-collection campaign (cf. [1]). The campaign collected contextual information from cell phones of participants.

Use-Case 1 (Nokia's Data-collection Campaign) *The data collected by a participant's phone is propagated into databases. For privacy reasons, data has first to be anonymized before it can be seen by the Nokia analytics department. Also users can*

delete their own data but deletions have to propagated in all databases. Data deletion/propagation is done by scripts.

Below the desired conditions:

script2 Only the script *script2* may delete data in db2

insert Data uploaded by the phone into db1 must be inserted into db2 within 30 hours after the upload, unless it has been deleted from db1 in the meantime.

delete Data may be deleted from db3 iff it has been deleted from db2 within the last minute.

The second use-case is based on the vision of smart medical environments where medical guidelines are carried out by smart medical devices, such as pumps. All of the taken actions are logged at a central authority. Online monitoring could be used to check all the logged actions for any violations. The use-case is based on a real-world medical guideline [4].

Use-Case 2 (Smart Medical Environment) *If a patient is scheduled for an infective endocarditis procedure, an antibiotic for prophylaxis should be administered by a smart pump before the procedure. If the antibiotic is not administered before the procedure, it may be administered up to 2 hours after the procedure. However, administration of the antibiotic after the procedure should be considered only when the patient did not receive the pre-procedure dose.*

Below the desired conditions:

before If a patient is scheduled for a procedure, an antibiotic for prophylaxis should be administed by a smart pump before the procedure.

2hrs If the antibiotic is not administered before the procedure, it may be administered up to 2 hours after the procedure. However, administration of the antibiotic after the procedure should be considered only when the patient did not receive the pre-procedure dose.

For this use-case, a number of executions are provided. In these executions, the number of patients/procedures varies.

References

[1] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. "The MonPoly Monitoring Tool". In: *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*. Ed. by Giles Reger and Klaus Havelund. Vol. 3. Kalpa Publications in Computing. EasyChair, 2017, pp. 19–28. URL: http://www.easychair.org/publications/paper/62MC.

[2] Klaus Havelund. *DejaVu*. [Online; accessed: 2019-05-17]. 2019. URL: https://github.com/havelund/dejavu.

[3] Klaus Havelund, Doron Peled, and Dogan Ulus. "DejaVu: A Monitoring Tool for First-Order Temporal Logic". In: *3rd Workshop on Monitoring and Testing of Cyber-Physical Systems, MT@CPSWeek 2018, Porto, Portugal, April 10, 2018*. IEEE, 2018, pp. 12–13. ISBN: 978-1-5386-6748-4. DOI: 10.1109/MT-CPS.2018.00013. URL: https://doi.org/10.1109/MT-CPS.2018.00013.

[4] Walter Wilson et al. "Prevention of Infective Endocarditis". In: *Circulation* 116.15 (2007), pp. 1736–1754. DOI: 10.1161/CIRCULATIONAHA.106.183095.

[5] Eugen Zalinescu. *MonPoly*. [Online; accessed: 2019-05-17]. 2019. URL: https://sourceforge.net/projects/monpoly/.