Semantic Rule Based Program Monitoring

Luke Tudor, Jing Sun
School of Computer Science
University of Auckland, New Zealand
Emails: ltud719@aucklanduni.ac.nz;
jing.sun@auckland.ac.nz

Hai Wang
School of Engineering and Applied Science
Aston University, United Kingdom
Email: h.wang10@@aston.ac.uk

Bingyang Wei
Department of Computer Science
Texas Christian University, United States
Email: b.wei@tcu.edu

***Abstract*—Program monitoring aims at making sure the functionalities of the software are always correctly performed during runtime. Semantic Web provides a context enriched framework for data representation and manipulation. This paper proposed the use of ontological rules and reasoning engines to monitor the dynamic behaviours of computer systems in handling of exceptional circumstances, both positive and negative, that occur at runtime within the software processes. A prototype framework was proposed on how to integrate the rule based monitoring technique together with the targeted system. To validate the proposed solution, a light control system case study together with the Unity game engine were used to develop a simulation environment for the evaluation purpose. Compared to existing solutions, the approach outlined can provide an effective software behavioural monitoring outcome.**

I. Introduction

In the past, research has been done into how software program monitors can be used to verify runtime correctness of an application by providing means of identifying errors when they occur. The goal of these monitoring systems is to provide a way to formally verify the correctness of the system in a way like automatic software testing, with the stipulation that this testing is done during the running of the system. Depending on the purpose of the program monitor, such as a debugging tool for developers, a logger or a system-wide message monitor, the exact way that these monitors are implemented changes with their purpose. These monitors can also provide many other features including precise error cause locations, methods to provide alerts about certain errors occurring or ways to recover from errors, e.g., by reverting a database to a previous state. However, these systems all have in common a way to specify what constitutes an error in the program and a way to provide feedback about that error, typically by logging or throwing an exception. Despite the potential application of program monitors in possibly increasing the overall quality of software, software program monitors are not commercially widespread and are not a part of typical software development workflow [1]. Recently, advances in the semantic web and ontologies have opened new possibilities for formal program modelling and verification. Additionally, these semantic web technologies provide powerful means for verifying constraints and conditions using rules and reasoning engines which could be useful for monitoring software program behaviour [2].

DOI reference number: 10.18293/SEKE2019-043

Since ontologies are a promising way to model software program data and thus monitor programs, the goal of this project is to investigate whether ontologies can be used in this way, and if they can, how best ontologies may be used for monitoring program data at runtime. Consequently, evaluation of how effective ontology-based monitoring is should be done. Because program monitors provide many different feature sets, levels of correctness and speed, amongst other factors. To determine the effectiveness of ontology-based monitoring, the best way to integrate a reasoning engine to the monitoring system should be explored, since the reasoning engine will be performing the runtime verification. Finally, the proposed system should be easy to use and more useful for developers, providing a possible alternative to conventional software testing methods.

From prior research, the monitoring approach used depends to a large extent on the intended use of the program monitor. Therefore, care must be taken to ensure that the monitor is fit for purpose. Since the intention of this project is to provide a monitor that can allow for error checking at program runtime at a single application level without developer interaction, such a system need not determine the origin of error states, nor does it require the implementation of sophisticated data recovery techniques. Additionally, although there exists within more recent research proposals for systems that integrate hardware into the monitoring of low fault-tolerant applications, to increase monitoring speed and protect against hardware faults [3], such a system would not be suitable for this purpose since the proposed system should be hardware agnostic for ease of use.

Examining past solutions, not all systems provide guaranteed fault detection. The most common cause is not mistakes in the proposed systems themselves, but in poor coverage provided by the predominantly control-flow driven monitoring techniques employed. Note that programs that have stricter runtime requirements with respect to real-time computing have more rigorous speed evaluations than those that do not, implying that for such systems, performance is of a greater concern than for systems where relatively slow human interaction comprises the bulk of the total process. Regarding the use of ontologies for modelling program data, there is evidence that such activity is possible. Other data formats such as XML are already used as intermediates for transferring data between different programs using middleware technology [4]. Since on-

tologies are designed as an expansion of such technology, with more sophisticated and standardised reasoning capabilities, it stands to reason that the functionality of semantic web and ontologies is a superset of the functionality of XML [5].

The objectives of this research are to design a system that can be used by software developers to monitor the runtime behaviour of programs and demonstrate how such a system can be used. To accomplish these objectives, the modelling of program data using ontologies needs to be explored and demonstrated for use in the ontology monitoring system. Also related to the modelling of program data is the way that rules and reasoning engines can be integrated into the system to provide the constraint checking necessary for this type of program monitoring to work. Furthermore, the proposed system should be easy to integrate into monitored programs, allow for specification of many different types of constraints on the data and be fast enough so that it could be realistically used in an actual software system.

This project aimed at achieving a usable, reliable and useful way to allow for integration of any software program into a runtime behavioural monitoring system and ways to verify correctness using many rules and constraints that can be easily and quickly changed to accommodate fluctuating requirements. Since this system focusses on providing a tool for developers and is difficult to evaluate independently of its use, a case study is proposed to test the monitoring system and to show how such monitoring can be extended to any generic program given an ontology structure. It should be noted that although the goal of this system is to simply provide a means for checking errors in programs, there exists the potential to implement some business logic in the defined rules on the ontology such that when reasoned about. These rules can make useful changes in the program data based on these functional requirements. Therefore, in the system demonstration, business logic inferences to maintain data consistency are demonstrated as they show off a superset of the potential uses of ontologies compared to the relatively simplistic logging of error states.

The rest of the paper is organised as follows. Section II presents the design of the system including software architecture decisions, technologies used, and the general methods used to integrate the program monitor with the monitored program and associated ontology. Section III presents the implementation of the system, including the construction of the monitoring system, the use of ontology and reasoners and an exploration of the case study as an example of a possible use case of the proposed system. In section IV, an evaluation of the monitoring system is discussed including the success of the testing methodology, comparison with previous program monitors, discussion of proposed system features, lessons learned and possible improvements. Finally, Section V concludes the contributions and discusses the future work.

II. System Design

The system proposed for integrating program monitoring into a piece of software is to create an instance of the program monitor within the monitored program using external APIs, and then using this monitor instance to update and receive updates from the ontology. This approach allows for easy integration with any monitored program by simply importing the relevant library whilst providing a high degree of control to the monitored program in deciding what properties are important within the monitored program. However, for the program monitor to work, an ontology and externally defined rules must be made available to the program monitor. This can be done within the monitored program by providing the location of these two files to the program monitor instance at construction time. Since updates from the program monitor should not be polled for by the monitored program, an interrupt style listener paradigm is used so that the monitored program can register an interest with the values of properties and provide code that runs when those updates are triggered. An overall data flow of the monitoring system in show in Figure 1.

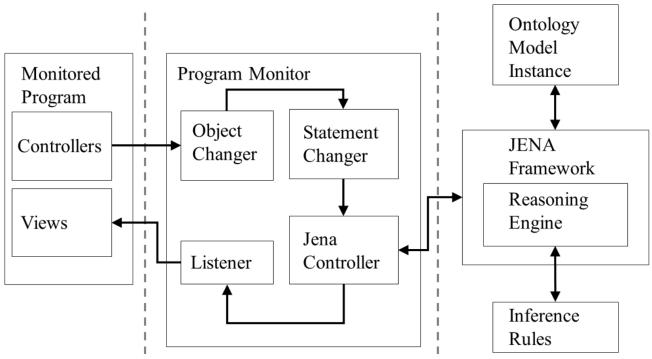

Fig. 1. Data flow through monitoring system

This call-back-like approach is known as the listener pattern and is an easy way to integrate program modules that do not need to be aware of when the other module runs given that updates are received eventually. For triggering updates, controllers are obtained from the program monitor instance to provide means to change property values from within the monitored program. The final part is the ontology model, which is updated firstly from the monitored program, then again by the reasoner if any updates are caused by the rules firing. This overall approach can be summarised as an implementation of the model-view-controller pattern (MVC) [9], which is often used for systems with a high degree of user interaction.

III. Implementation Detail

Conceptually, the monitoring system lies between the monitored program and its corresponding ontology; the monitoring itself works by checking the entire ontology against all the rules every time the Jena API is called by a property changer to update the given ontology. The timing of the rules firing is managed by the Jena framework, however, since Jena is open source, the rules engine could be made to run at different times. Since the controller objects in the monitored program use the call-back principle to specify changes, it is easy to change value types and access core Jena functionality with little additional work.

These statement changers or monitor controllers require direct access to the ontology model and must be requested from the program monitor instance rather than be constructed directly. At construction time, the full URL of the resource and property to be updated is supplied by the controller to the program monitor instance to obtain the resource and property Jena objects associated with those URLs. However, since the ontology objects for each entity change values, these property values must be located each time the object pointed to is updated. After an update is made to the ontology, all listeners are notified synchronously of the current value of the property objects that they are listening for. As with the controllers, a reference cannot be kept for a listened object, so each listener supplies URLs for the resource and property to use to retrieve listened for objects each time an update occurs. These updates are then passed to the listeners using the call-back-like method, invoking a change in the monitored program somewhere. After all listeners are notified, control is passed back to the monitored program for the next update.

Before the monitoring system should be integrated into the target system, consideration must be made for the construction of the ontology and associated rules. The ontology should be constructed such that all possible conditions that might be reasonably monitored are represented in an externally logical form and such that the ontology can provide an accurate representation of the important data in the program. This means that each object in the program that represents something in the physical world should be represented in the ontology; more specifically, objects that have value outside of necessary software development usage within programs, like array lists or hash tables would only appear in the monitored program. In other words, an ontology should represent the context schema of a program. The classes of the ontology should represent the types of object to be modelled, the entities should represent the instances of those objects within the program and the ontology properties should represent the fields or attributes that are of interest within these entities. From these parts, a complete model of the ontology of a system can be constructed. For this project, OWL [5] ontology reasoning was used. Jena supports OWL DL (Description Logic) specifically, which allows more powerful reasoning than would be provided by a less expressive ontology language such as RDF. Notably, when using the proposed monitoring system, the ontology should be modified by an external tool such as Protege [10] which is designed for easy editing and analysis of ontologies.

IV. Case Study and Evaluation

A. The Light Control System

To demonstrate the proposed system, a case study is required that demonstrates how monitored programs can be integrated with the monitoring system and how effective the proposed system is at monitoring programs. A building management system was chosen for the demonstration since building management systems are typically well-defined, contain many complex rules and constraints and are suitable for demonstration in an interactive environment such as a game engine. Additionally, such a simulation could be feasibly extended to a physical sensor network if the associated simulation is successful at capturing all the necessary functionality in a similar way. The basis for the chosen building management system is provided in [12]. This description provides a high level of detail about a typical smart building complex with constraints related to context-aware features such as the temperature and light intensity controls [13].

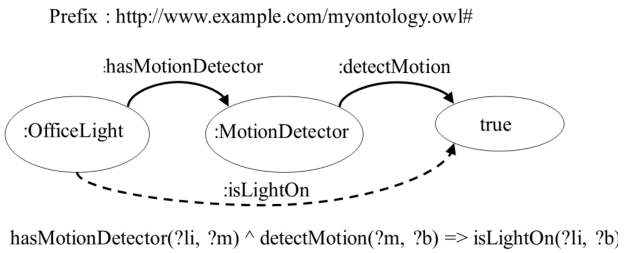

Fig. 2. Simplified ontology example.

In this example, there are at least three entities which may be formalised by an ontology, i.e., the motion detector, the light and the room which contains both these objects. Therefore, this interaction can be represented by a SWRL [11] rule which updates the '?light on?' property to false whenever the motion detector in the same room has the '?motion detected?' property evaluate to false. Figure 2 shows a simpler version of this scenario (without the connecting room entity) to illustrate how rules can be used to infer relationships using a reasoning engine. Extending this scenario to every room in the building with a motion detector allows the same rule to be used in each case.

B. Evaluations

For validating the monitoring system, 10 rules of varying complexity and approach were constructed to show that when updates relevant to each rule are received, that rule is fired, producing some change in the system ontology. Of these 10 rules, each was tested within the simulation environment and all rules were found to fire when expected and with the expected results. These rules were constructed such that every rule possible was paired with a rule that fired under opposite conditions and produced the opposite result. The motivation behind this rule construction methodology was to ensure that in a test environment, that the updates and rule changes were repeatable. The rules that did not have an opposite satisfied this condition of repeatability by including the object part of the triple condition within the rules as a wildcard or free variable, using this variable input to compute a variable output. This meant that some rules could contain the same functionality as two rules in the case of a Boolean literal object, or that rules which used integers could contain the same functionality as an infinite number of more specific rules. This behaviour is of interest since assertions are typically static during runtime, and different cases of assertions cannot be compressed into a single assertion. Thus, it seems that rules can sometimes provide more powerful condition checking than assertions.

For the testing environment, rooms were constructed within a simplistic mock building within the game engine so that each rule could be tested in isolation and without interference as many times as desired. This approach is a generalisation of typical testing techniques, with the main differences being that rules could be tested systematically by following a defined path through the different rooms, whilst allowing for rules to be fired at any time and with any frequency, more closely mirroring actual human interaction with a system. Although this style of testing is less automatic than traditional testing means, errors associated with timing and user experience are much easier to identify if each test is run eventually. In addition to the freeform testing provided by the simulation environment, a JUnit test suite was used early in the project lifecycle to evaluate the reasoning engine before the simulation environment had been completed. As with the simulation environment, all rules were fired when expected and produced the expected results.

A benefit of using the system is that ontologies can easily be reused and transferred between multiple formats, this contrasts with other program monitoring approaches which are more specific to each monitored program which may require more redesign work for slightly different applications. Another benefit compared to assertion-based systems is that the code is not cluttered with annotations that obscure the code intent or need extensive work to change if the monitored behaviour changes due to new requirements. This constraint checking work is delegated to the rules file, which provides a more cohesive interface for changing checked behaviour. Compared to the more common control-flow based monitoring, the proposed system is simpler to understand conceptually and easier to reason about from a monitoring perspective. This is because checking the control flow through a conditional or function often assumes some flow higher up in the control of the program, which can make it complicated to get a total view of the system status.

The proposed system can be compared to other program monitoring approaches based on each system's relative feature set. The features of several other program monitors are summarised in this paper. In comparison to systems that require specialised hardware, such as [3], this approach provides guaranteed correctness, given that the rules and ontology are constructed correctly without the hassle of customised hardware. Software only monitors to insert assertions automatically have been proposed [7], however it and other automatic assertion generating programs do not guarantee correctness unlike [3]. Systems that use constraints to monitor program execution also exist [6], however, these systems can be too heavyweight for smaller projects and do not provide the benefits of using ontologies as discussed previously. Assertion based monitors like [8] can also be used, but the main disadvantage of assertions to monitor control flow, is that control flow monitoring can become too complex to easily modify and rules must be changed from within each software module monitored. Although the monitoring system described in this paper solves the previously discussed problems, it is not without disadvantages, the most notable being the additional work required to make and maintain the ontology and the performance of the system.

V. Conclusion

Program monitoring aims at ensuring functionalities of the software system are always correctly performed during runtime. This paper demonstrates not only that semantic ontology and its reasoning engines can be integrated with software applications to allow for rule-based monitoring, but also outlines a method for doing so. Additionally, the effectiveness of such a tool was evaluated with respect to how well a reasoning engine could determine errors in software and how useful the tool would be for software developers. This project has found that it is not only possible to create a powerful and flexible tool to monitor programs using rules and ontologies, but also the tool can be easily integrated with existing applications. These contributions were gathered based on implementing and testing a semi-realistic case study integrated with a game engine simulation environment to provide real-time feedback on ontology updates and rule firing. In addition, comparisons to related work were conducted with useful evaluations. In the future, a feature that would be greatly increase usability would be the ability to convert rules from more common languages such as SWRL into the Jena specific rule format.

References

[1] A. Bertolino,*Software Testing Research: Achievements, Challenges, Dreams*, Future of Software Engineering (FOSE '07), Minneapolis, MN, 2007, pp. 85-103.

[2] Kishore, Rajiv, Ramesh, Ram (Eds.), *ONTOLOGIES: A Handbook of Principles, Concepts and Applications in Information Systems*, Boston, MA: Springer US, 2007.

[3] J.R. Azambuja, M. Altieri, J. Becker and F.L. Kastensmidt, *HETA: Hybrid Error-Detection Technique Using Assertions*, in IEEE Transactions on Nuclear Science, vol. 60, no. 4, pp. 2805-2812, Aug. 2013.

[4] Steve Graham, Doug Davis, Simeon Simeonov, Glen Daniels, et al., *Building web services with Java*, Que Publishing, June 28, 2004.

[5] W3C Recommendation 10 February 2004, *OWL Web Ontology Language Overview*, 2004.

[6] W.N. Robinson, *Implementing Rule-Based Monitors within a Framework for Continuous Requirements Monitoring*, Proceedings of the 38th Annual Hawaii International Conference on System Sciences, Big Island, HI, USA, 2005, pp. 188a-188a.

[7] N. Oh, P.P. Shirvani and E.J. McCluskey, *Control-flow checking by software signatures*, in IEEE Transactions on Reliability, vol. 51, no. 1, pp. 111-122, March 2002.

[8] D. Bartetzko, C. Fischer, M. Mller and H. Wehrheim, *Jass - Java with Assertions*, Electronic Notes in Theoretical Computer Science, vol. 55, pp. 103-117, Oct. 2001.

[9] G.E. Krasner and S.T. Pope, *A description of the model-view-controller user interface paradigm in the smalltalk-80 system*, Journal of Object Oriented Programming, vol. 1, pp. 26-49, 1988.

[10] N.F. Noy, M. Sintek, S. Decker, M. Crubezy, R.W. Fergerson and M.A. Musen, *Creating Semantic Web contents with Protege-2000*, in IEEE Intelligent Systems, vol. 16, no. 2, pp. 60-71, March-April 2001.

[11] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean, *SWRL: A semantic web rule language combining OWL and RuleML*, W3C Member Submission, vol. 21, pp. 79, 2004.

[12] S. Queins, M. Becker, M. Kronenburg, C. Peper, R. Merz and J. Schfer, *The Light Control Case Study: Problem Description*, J.UCS: The Journal of Universal Computer Science, vol. 6, 2000.

[13] J. Sun, H. H. Wang and H. Gu, *Semantic Enabled Sensor Network Design*, in proceedings of 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011), pages 179-184, July 7-9, 2011.