Inspection of distributed and composed systems

Wouter De Borger, Bert Lagaisse and Wouter Joosen
Distrinet, Department of Computer Science, KULeuven, Belgium
wouter.deborger@cs.kuleuven.be

ABSTRACT
Middleware enables developers to build more complex applications by shielding them from the complexity of the underlying platform and environment. However, when inspecting software at run-time, the abstractions offered by the middleware are no longer visible.

To support advanced control of middleware based systems, in a dynamic cloud environment, my goal is to provide the proper tools to enable inspection of complex, composed systems, in terms of the most appropriate abstractions.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging; D.2.8 [Software Engineering]: Metrics

Keywords
Middleware, Monitoring, Debugging, Model transformation, Complex Event Processing

1. INTRODUCTION
With the advancement of middleware research, more complex building blocks become available to a wider audience. Complex components such as rules engines, business process engines, message routers and distributed caches are already included in mainstream middlewares. This allows more people to build complex, heterogeneous systems, composed out of different languages.

At the same time, the move towards cloud computing brings the inherent challenges of distributed systems to a wider audience. The cloud environment is only usable due to the associated middleware support. Middleware abstractions shield developers from the inherent complexity and allow them to focus on functionality.

However, once the system is deployed, the abstractions are lost. From an outside perspective, the middleware can no longer distinguished from the application. Middleware and application code are composed into a blob in which the full complexity of the environment is exposed. All the abstractions that enabled and supported development are replaced by a complex synthetic run-time structure.

The focus of my work is to provide the proper tools to enable inspection of such complex, composed systems. In the spirit of Bracha et al.’s mirroring principles [5], we aim to provide tools that can represent a system in terms of the abstractions most appropriate for the task at hand.

More specifically, these principles are:

- Encapsulation: meta-level facilities must encapsulate their implementation. Their interface and implementation should not be part of the base-level, to allow independent change of the base- and meta-level.
- Stratification: meta-level facilities must be separated from base-level functionality.
- Structural correspondence: the structure of meta-level facilities should correspond to the structure of the language they reflect on.
- Temporal correspondence: meta-level APIs should be layered in order to distinguish between static and dynamic properties of the system.

To achieve this goal, we propose a project with three phases: exploring the state-of-the-art, extending the state-of-the-art and validation. We are currently in the second phase.

2. EXPLORING THE STATE-OF-THE-ART
Inspection requirements change throughout the software life-cycle. At development time, tools such as debuggers and profilers are used to provide very detailed inspection. In production, security enforcement, monitoring and performance analysis tools are used. When a system has crashed or is compromised, post-mortem analysis tools are used to restore state and find possible irregularities. Each of these tools has specific requirements and specific implementation strategies.

We distinguish three classes of tools: interactive, continuous and detailed tools.

- Interactive tools, such as debuggers, are used to interactively explore the run-time structure. These tools are not used in production and are thus allowed to have a large influence on the execution. This kind of tool is external to the application (part of the execution environment). The requirement for an interactive
and intuitive inspection model often results in surprisingly complex tools.

- Continuous tools, such as monitors, monitor a limited set of properties, but do so continuously. This kind of tool must not interfere with normal operation, as they are used in production. The non-interference requirement forces these tools to be simple and limits the amount and type of information that can be collected.

- Detailed tools, such as performance analyzers, make detailed inquiries about in-production systems. They are used in production environments and must not interfere with normal operation, but they are not continuous. They are dynamically added when required. This allows them more freedom in what to inspect, but their dynamic nature makes them complex.

To explore this diversity, we built two innovative prototypes in the context of Aspect Oriented Software Development (AOSD), each representing a different class of inspection tools. First we built AODA, the aspect oriented debugging architecture [9]. AODA is a debugger for AspectJ [1] like languages, which provides an abstraction over the complexity of such languages. It is an interactive tool, performing complex data transformations to translate the run-time state back into the original programming abstractions. It supports both the AspectJ language and also the JBoss AOP middleware [6].

Second, we built AOPS, the aspect oriented programming security model [8]. AOPS is a run-time permission system for AO languages. It maintains a simple model of the program state to support policy enforcement. As a continuous tool its implementation is simple and efficient. However, while the security model is simple, it is not as rich or intuitive as AODA.

Aside from these experiments, we also explored the related work from the fields of debugging, monitoring, reflection, run-time security, model driven development and complex event processing. The related work confirmed that the dominant research challenge in the field of inspection is not information extraction, but information aggregation. For information extraction, many solutions exist and are already actively used. However, when multiple sources of information are present or when the information has an abstraction mismatch, no middleware support exists that can go beyond simple statistical aggregation. In Traceability between run-time and development time abstractions [19], we elaborate on the importance of complex information aggregation to allow a system to be presented in terms of the most useful abstractions.

To summarize the relevant related work very briefly:

- **Debugging.** Debuggers extract information from their target by specific and often complex protocols. This information is then transformed into an understandable and more manageable format. When building and surveying debuggers, it became apparent that these transformation components are quite complex. Also, there is currently no body of research about such transformations. This limits the ability of debuggers to handle more complex abstractions. GDB for example [10], is an advanced multi-platform debugger, but still lacks the ability to decode the heap of C programs. Currently efforts are on their way to improve the modularization of the transformation components, to allow the heap to be decoded [14].

- **Monitoring.** The research around monitoring is focused on abstraction [3, 2, 15]. The need to correlate monitoring events has been one of the main drivers behind the research into complex event processing and rules engines [13, 18]. As such, complex event aggregation is quite well supported.

- **MDD.** The model transformation approaches developed in the context of MDD provide technology similar to complex event processing, aimed at the aggregation of state instead of events. These approaches are theoretically capable of state aggregation for inspection, but their current implementations make this practically impossible [7, 12, 16, 17, 11].

3. **EXTENDING THE STATE-OF-THE-ART**

From the state-of-the-art it is clear that two important components are missing to allow inspection information to be aggregated: 1) a convenient means for state transformation, 2) a convenient means for hybrid transformations, that combine both event- and state transformations.

In our current work “A generic solution for agile run-time inspection middleware” [4], we have addressed the lack of support for state transformations. We have adapted the QVT transformation language so that it can be used for distributed inspection. We have developed a prototype that supports inspection of RMI applications in terms of RMI instead of Java. We extended the Java debugger with distributed stack traces and remote objects. This research demonstrates how inspection components can be derived from a declarative specification. Even when the specification contains redundancy, an efficient implementation can be derived. This allows more modeling freedom and enables easier development.

In the future, we plan to integrate this transformation approach with an event aggregation system, to allow automatic generation of hybrid inspection components. Our current approach describes how abstractions relate to each other. It defines which facts are used to derive other facts. Events on the other hand signal the change of facts. When an event is received, this can be propagated through the derivations. i.e. When an event signals the change of a fact in a lower abstraction, the transformation can be used to propagate this change to the higher abstractions. This allows the inspection information to be updated automatically and new high-level events to be generated.

4. **INTEGRATION AND VALIDATION**

In this final phase, we will integrate the previous efforts and validate them. We aim to provide an inspection framework that can control multiple sources of information and use them to create an integrated view of the system. We will apply this platform to a case study to measure its performance and usability.

5. **REFERENCES**


