Applying Autonomic Diagnosis at Samsung Electronics

Paulo Casanova, Bradley Schmerl and David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
{paulo.casanova,garlan,schmerl}@cs.cmu.edu

Rui Abreu
University of Porto
Porto, Portugal
rui@computer.org

Jungsik Ahn
Samsung Electronics
Republic of Korea
jungsik.ahn@samsung.com

September 30, 2013
CMU-ISR-13-111

Abstract

An increasingly essential aspect of many critical software systems is the ability to quickly diagnose and locate faults so that appropriate corrective measures can be taken. Large, complex software systems fail unpredictably and pinpointing the source of the failure is a challenging task. In this paper we explore how our recently developed technique for automatic diagnosis performs in the automatic detection of failures and fault localization in a critical manufacturing control system of Samsung Electronics, where failures can result in large financial and schedule losses. We show how our approach scales to such systems to diagnose intermittent faults, connectivity problems, protocol violations, and timing failures. We propose a set of measures of accuracy and performance that can be used to evaluate run-time diagnosis. We present lessons learned from this work including how instrumentation limitations may impair diagnostic accuracy: without overcoming these, there is a limit to the kinds of faults that can be detected.

Acknowlegements

This work was supported in part by the National Science Foundation under Grant CNS 1116848, the Army Research Office under Award No. W911NF-09-1-0273, the Office of Naval Research under Grant N000141310401 and Samsung Electronics. Furthermore we would like to thank Jungsik Ahn, of Samsung Electronics, who helped us define the simulator described herein.

Chapter 1

Introduction

Product manufacture, notably technology manufacture, is highly automated, involving critical software to schedule, control, and monitor the manufacturing process. The large volume of items produced, the high competitiveness in manufacturing costs, and global supply chains make failures in this industrial setting costly, leading to schedule overruns and large financial losses [9, 14].

The inherent complexity in the manufacturing process, exacerbated by extremely optimized production processes, and the high volume of production makes identifying faults in the underlying running software difficult [12]. Software systems controlling the manufacturing process communicate by sending a large number of messages between equipment and the controlling software to coordinate manufacturing, and involve little human intervention. Although hardware faults (i.e., equipment failure) may also lead to software faults, in this paper we primarily address failures triggered by software faults.

While this software system works most of the time, intermittent faults can cause cascading errors that can delay or halt manufacturing. In such a case, the software typically needs to be rebooted, causing further delays. Furthermore, isolating which parts of the system caused the initial problem is a difficult process, where developers manually examine volumes of log files to see if they can determine why the problems occurred.

Identifying failures in a running system is challenging [4, 7, 10, 11, 16]. First, while failures commonly manifest themselves at a high level of abstraction, like degradation of quality of service below minimum thresholds or violation of high-level protocols, monitoring is usually performed at a lower level and we need to bridge the gap between these two levels. Second, these systems perform multiple activities concurrently. These activities, even if they are unrelated to each other, will appear mixed to an observer and it is necessary to split them in order to check their correctness.

Performing diagnosis is also challenging, even assuming that failures are correctly identified. First, there can be multiple possible explanations for a failure. For example, a system slowdown may be caused by a slow database, a component failure or a faulty network connection. Second, diagnosis has to be performed within a useful time frame. For example, if diagnosis takes too long to detect that an equipment is malfunctioning, the production process may already have been compromised.

In this paper, we report on our experience applying our run-time fault diagnosis approach [6] to dynamically and automatically diagnose and locate faults for a chip manufacturing process for Samsung Electronics. The criticality of the control software prevents us from performing monitoring and diagnosis in the real system. Also, lack of factory equipment makes using the real software impossible. Therefore, we developed a simulator mimicking the behavior of the Samsung Electronics' manufacturing control system, which has been validated by Samsung Electronics.

In previous work we have tested our approach successfully on several research prototype software systems, reported in [6, 5]. In this paper we discuss how the technique works in a real-world setting (albeit simulated), with the scale and problems that exist in that system. In particular, we validate that our approach to autonomic monitoring and diagnosis (a) *accurately* identifies the fault of a problem by observing the run-time behavior; and (b) achieves the desired *performance* without violating quality attributes of the system. Furthermore, in our previous work we have argued that:

- The specifications of system behavior required for our approach were an intuitive concept for system designers.
- These computations could be reused within systems of the same architectural family.
- Defining correctness criteria for computations is simple, being directly derived from business requirements and quality attributes, and in terms familiar to system designers.
- The statistical analysis provided by the diagnostic algorithm can pinpoint the source of the fault accurately.

In this paper we analyze how our algorithms matched the challenges and how they fulfill the claims just outlined. We further propose two metrics to evaluate runtime diagnosis algorithms. Previous design-time fault localization techniques have only considered diagnostic accuracy as evaluation metric. We argue that at run-time one is not only concerned with diagnostic accuracy but also with the performance of the algorithm. Hence, we propose this two metrics to evaluate the performance of run-time fault localization techniques. Finally, we summarize lessons learned, which we believe to be applicable in other run-time diagnosis systems.

This paper is organized as follows: in section 2 we provide a description of Samsung Electronics' system we're targeting. Section 3 describes our approach to autonomic diagnosis and how it was applied in Samsung Electronics. In section 4 we propose metrics for evaluation of run-time diagnosis algorithms. In section 5 we provide the results of our case study in a set of scenarios mimicking problems detected at Samsung Electronics. In section 6 we list lessons learned that can be useful for future systems. Section 7 summarizes our finds, discusses threats to validity and future work.

Chapter 2

Target system

Samsung Electronics' manufacturing system is a large-scale industrial control system responsible for manufacturing control of semiconductors[1]. The system controls the stages of wafer manufacture, deciding to which equipment wafers are sent for processing. Furthermore, the software tracks of not only the lots being manufactured, but also the equipment used: which equipment is allocated, whether and what it is processing, what its output quality is, and so on.

The system is divided into subsystems, which perform specific tasks related to the manufacturing process. For example, the MOS (Manufacturing Operating System) subsystem controls the stages of the manufacturing process, and the ADS (Automatic Dispatch System) subsystem performs equipment scheduling, deciding which equipment is the best to perform a step in the manufacturing process.

Each of these subsystem is built by multiple concurrent processes that provide various manufacturing services, load distribution, and fault tolerance. These are connected through an event bus, which mediates event exchange amongst them. A subset of the system's architecture is abstractly depicted in Figure 2.1. Due to confidentiality reasons, specific details about system implementation cannot be reported. The total message throughput in all buses combined is around 2000 events per second.

The systems communicate with each other exchanging messages according to several predefined protocols. One such protocol is the *track-in* (`TKIN`) protocol. This protocol is used before a wafer lot is sent to another stage of processing. It determines what equipment the wafers should be sent to, performs validation operations and does some housekeeping, like ensuring that the equipment for processing the next steps are available and that no scheduling and quality constraints are violated.

Several factors make this system complex:

1. The `TKIN` protocol contains more than just message exchange between components: it also involves databases accessed through standard proto-

[1] http://www.samsung.com/global/business/semiconductor/

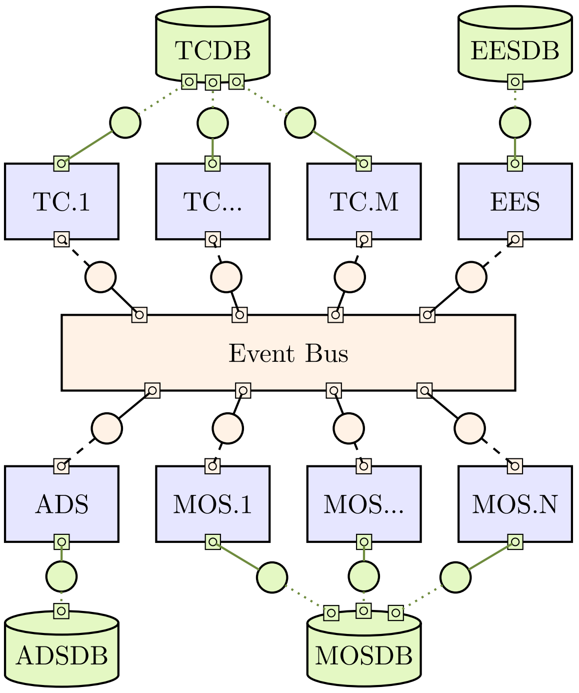

Figure 2.1: High-level architectural view of the manufacturing system at Samsung Electronics. For simplicity, only the `MOS` and `TC` systems are shown decomposed.

cols. This means that observation of the event bus will only yield partial information.

2. Several components of the same type exist, but only one will receive each message although it is not known which. This makes tracking messages harder as we don't know what the destination of the message will be: all possible destinations have to monitored and searched for a match.

3. Multiple TKIN protocols are occuring in parallel and they may be executed with different timings. For example, the `MOS` may send two `BEST_EQP` messages to the same `ADS` which returns two `EQP` responses whose contents have to be analyzed to identify which response matches which request.

4. More messages circulate through the bus between these components to address other protocols besides the `TKIN` protocol. This means that we will see spurious messages in the bus.

2.1 Operation issues

While `TKIN` protocols are executed correctly most of the time, once in a while problems arise that cause failures in the system. These problems vary, but some of the typical problems are messages being lost (or not sent at all), messages sent too late, or unexpected messages being sent. Other problems are database performance slowdowns, which affect overall system performance.

Such problems can have a significant impact on the manufacturing process as they can lead to stalling lots and unneccesary equipment reservations. Given the sheer volume of messages being exchanged it is not possible for human operators to even realize that a problem has occurred until later when more serious consequences become visible. Also, given the independent development of all the systems involved, it is difficult for the system developers to figure out what the problem is and where it is located.

Yet another difficulty is that these problems, although serious, are rare - and therefore are difficult to diagnose. Although the system operates with around 2000 messages exchanged per second, it generally functions correctly for many weeks in a row before any failure is detected. This sets up a scenario in which millions of observations are performed before a single failure can be identified.

2.2 Simulating the target system

There are several barriers to developing a diagnosis system for systems such as the Samsung Electronics' manufacturing control systems. The criticality of such systems makes testing on the production system unacceptable; the size of the system and its need to work connected to factory equipment makes it impossible to run it in a different environment. Also, the rareness and unpredictability of the faults make testing a diagnosis system difficult.

We addressed these issues by creating a simulator of Samsung Electronics' production system and performing diagnosis in the simulator. This simulator was engineered with the help of Samsung Electronics to produce a TKIN protocol resembling the real one and to allow manual control of faults for testing purposes.

The TKIN protocol is described using a message sequence diagram in Figure 2.2. The messages in the sequence diagram correspond to types of events sent: for example, BEST_EQP is a request for an equipment to process a wafer lot and EQP is the event with the ADS's decision on the equipment to perform the process step.

The simulated system does not, naturally, control any equipment. Rather, it generates messages that conform to the protocol, both in terms of their types and also in their timing. This simulator allows us to develop test scenarios in which we arrange for specific faults to happen, check whether they are diagnosed correctly, and have a credible expectation that it can be successfully transitioned into Samsung Electronics' production system.

The simulated system is similar to the one in Figure 2.1, but it has only two MOS components, one event bus, and two TC components. This is in contrast with the real system which includes more than 20 of each type. However, because the computation performed by our individual components is much simpler than the real one, they are able to generate a much higher message rate than their real counterparts yielding a message rate closer to the real system. Consequently, our smaller number of components is actually simulating a higher number of real system components from a message exchange perspective.

The system starts multiple TKIN protocols at random time intervals. Components have random processing delays and a single TKIN protocol takes around 1 minute to complete, similar to the TKIN protocol in the real system. The rate at which TKIN protocols are initiated can be used to control the level of concurrency.

Both MOS components work in fault-tolerance mode: either MOS.1 or MOS.2 will receive messages addressed to the MOS, but not both. They maintain a "keep-alive" mechanism, allowing one to take over if the other fails to respond. The TC components work in "load-distribution" mode: each request will be forwarded to *either* TC.1 or TC.2.

Implementing fault tolerance in our simulator was done by creating a synthetic component (MOS.S) that acts like the MOS for all other components (except MOS.1 to MOS.2). It will forward through the event bus any message to either MOS.1 or MOS.2, depending on which component is active. Load distribution is implemented similarly using a synthetic TC.S component. This mimicks the way the event bus system used at Samsung handles load balancing and fault tolerance.

Although we did not have access to the actual factory control software, we were careful to work with Samsung Electronics engineers to ensure that the simulator we developed was faithful to the real system in the following ways:

- event timings were designed to produce delays close to the real system;

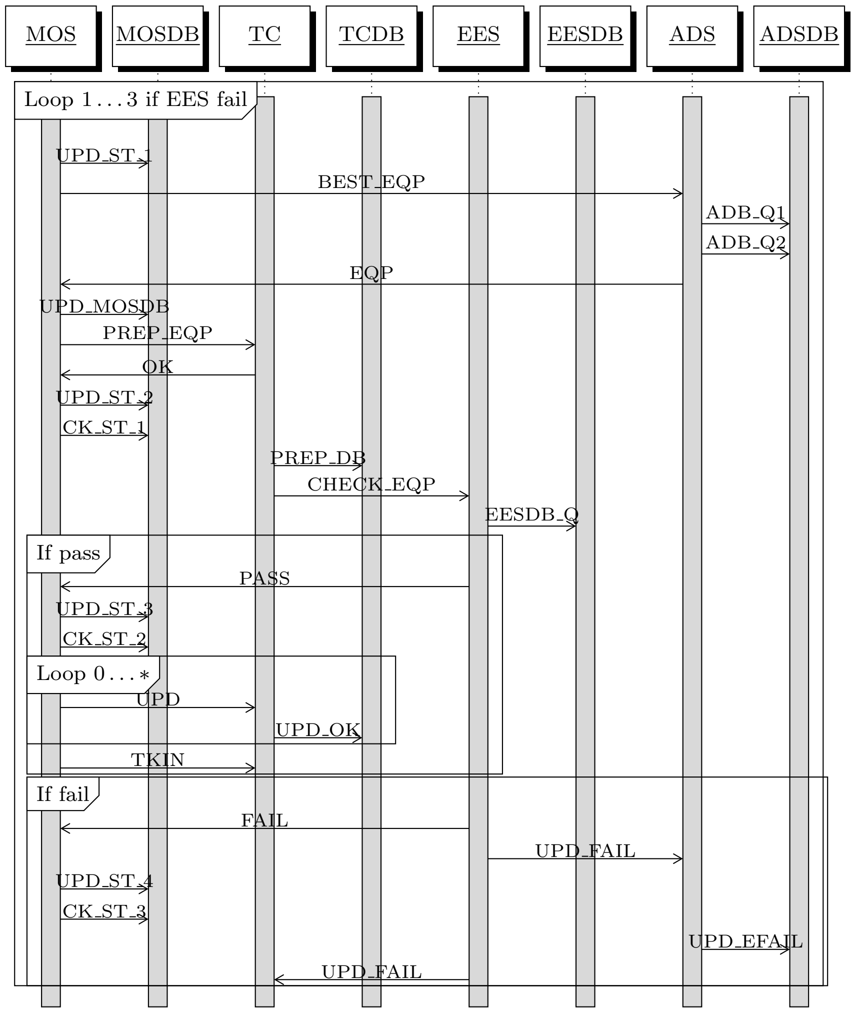

Figure 2.2: Simulated TKIN protocol.

- the protocol was designed according to Samsung Electronics' specifications;
- the faults that were injected (see Section 5) were typical of real problems experienced by Samsung Electronics and which are hard to locate in the real system.

We are therefore reasonably confident that the results reported in this paper will apply when Samsung Electronics transitions our approach to the real system.

Chapter 3

Approach

This section details our approach to automatic monitoring and fault localization, as well as how we applied it to the Samsung Electronics' system.

3.1 Overview

A detailed description of our approach for autonomic diagnosis is described in [5, 6]; see Figure 3.1 for an overview. It consists, at a high level of abstraction, of defining a behavior model (equivalent, to some extent, to the `TKIN` protocol described in Figure 2.2) over the system's architecture (such as the one in Figure 2.1) and monitoring the system to identify patterns that match to the model.

These behavior specifications are high-level computations[1] that occur at the architecture level. These high-level computations are not directly observable in the system. Instead, low-level events such as a message sent from the `EES` to the `ADS` with type `UPD_FAIL` for equipment lot `X.445`, can usually be extracted through instrumentation. These low-level events are then combined by the *Recognizer* using a *Behavior Model* (see figure 3.1), to form the high-level computations.

The high-level computations are then analyzed by an *Oracle* to check whether they represent correct or incorrect behaviors, according to some predefined *Correctness Criteria*. Correctness is derived from the business rules that define the system's functional requirements and quality attributes. For example, an incorrect computation can occur because a protocol did not complete successfully, or it completed, but without satisfying desired performance requirements. This information is sent to a *Fault Localizer* which will compute health estimates for the component and connectors in the system.

The output of the fault localizer is a set of fault candidates ranked by the probability of being the explanation of the observed failures [2]. The fault

[1]We termed them *transactions* in [5, 6], but use *computations* to avoid confusion with database transactions

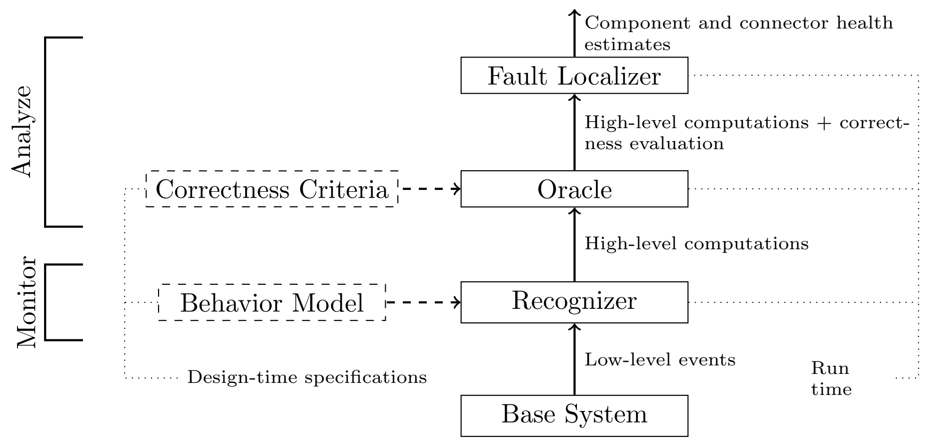

Figure 3.1: Structure of the diagnosis system.

localizer gathers information until it is able to produce an accurate report based on a limit on entropy [15]. Borrowed from information theory, entropy is a measure of the uncertainty in the ranking produced by the diagnostic algorithm. This allows us to dynamically adjust the number of computations we need to observe before being able to accurately diagnose a fault.

Having a set of computations over the architecture classified as correct and incorrect allows us to use static-time algorithms used for fault localization, such as the one in [2]. This algorithm was initially designed to localize faults in source code in the presence of a set of successful and unsuccessful runs of test cases (i.e., at development time, during the debugging phase of the software development life-cycle). We apply them at runtime, using observed run-time computations as "test cases", and localize the faults to the respective components in the system's architecture.

The essence of the technique is identifying which components contributed to which computations, determining the possible combinations of failures in components that could explain the visible faults. These combinations of components are then probabilistically ranked according to the likelihood of explaining the visible faults. Being a technique that reasons over information about run-time behavior, the diagnostic accuracy increases with the number of observations.

The key ideas of our approach are:

- System designers reason about systems using high-level computations like the `TKIN` protocol, which are meaningful from a *business* perspective and whose structure and design is driven by the system's requirements.

- It is possible to classify these high-level computations as either *correct* or *incorrect* behavior. Since these computations are meaningful from a business perspective, their correctness is usually defined directly in terms of system requirements. For example, a `TKIN` protocol which halts half way through is considered *incorrect behavior*. Also, a TKIN protocol which takes more time to complete than the established maximum is also *incorrect behavior*.

- High-level computations of a system cannot generally be observed directly in a system, but can be detected by observing lower-level events. For example, it is not possible to directly see the TKIN protocol, as a whole, in the system. But we can see individual messages sent from components to other components. These individual messages (which *are* directly observable) allow us to reconstruct the higher-level behavior, the TKIN.
- Run-time observation of both correct and incorrect behavior can be fed into design-time fault localization algorithms as if each run-time observation was the execution of a test case.

3.2 Architecture

An illustration of the architecture of the approach is presented in Figure 3.2. To achieve a high level of parallelism that supports scalability, each individual step in the recognition of the computation is done in its own individual recognizer. So, matching the BEST_EQP sent by the MOS to the EQP received by the ADS is done by one recognizer but matching the resulting pair with the respective PREP_EQP sent by the MOS is done by another recognizer that executes concurrently with the first, in an independent process. The same structure applied to the Oracle: each individual computation type is handled by an individual Oracle and all individual Oracles are processing in parallel. All individual Recognizers and individual Oracles communicate through a private event bus system that can scale up to thousands of messages per second.

The recognizer and oracle of Figure 3.1 are therefore broken down into multiple independent processes that communicate through an event bus. This event bus is *not* the same as the one used for the target system.

3.3 Behavior specification language

To apply our diagnosis framework to a system, we have to provide both a behavior model of the system and the correctness criteria (see Figure 3.1), which are used as inputs by the Recognizer and the Oracle.

The behavior model describes how higher-level computations are built from lower-lever computations. This is done by declaring individual recognizers that will be instantiated at runtime. The structure of a recognizer is represented in Listing 3.1.

```
/* Declares the recognizer "name". The formal parameters represent the
     computation types this recognizer will use to recognizer a higher-level
     computation. */
recognizer name(a0 : a0_type, ...) {
/* The invariant contains a first-order predicate logic statement over the
     arguments that recognized computations must maintain. */
```

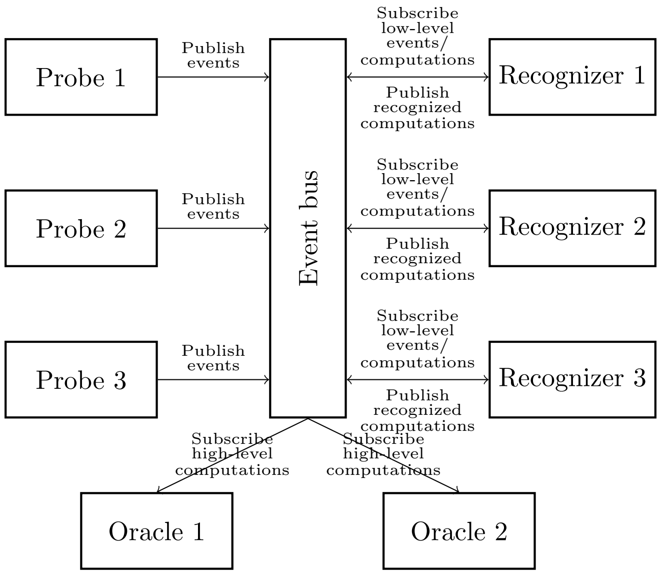
Probe 1
Probe 2
Probe 3
Publish events
Event bus
Subscribe low-level events/ computations
Publish recognized computations
Recognizer 1
Recognizer 2
Recognizer 3
Subscribe high-level computations
Oracle 1
Oracle 2

Figure 3.2: Architecture of the diagnosis system. Each box represents an independent process.

```
    invariant ... ;
/* The emit clause (more than one may be present) defines what higher-level
    computations are recognized. */
    emit ... ;
/* Alternatively, a emit fail clause says that we explicitly identified
    incorrect behavior. */
    emit fail;
}
```

Listing 3.1: Structure of a recognizer.

An important aspect of recognition is how to treat events that do not fit into any pattern. We adopted a conservative approach: we assumed our knowledge of how the system works is limited and, therefore, more computations can be detected that we do not know about. Unmatched events are ignored.

In some cases we know that some events must always be matched according to some pattern, and the absence of a match signals a failure. For example, a request for an equipment without a response indicates a failure. Our approach supports detecting these situations by specifying recognizers that detect illegal computations. These recognizers, which make use of first-order logic's existential quantifiers, can detect that there is an event of type `X` for which there is no match with a required event. They use the `emit fail` clause to report that incorrect behavior has been detected.

The correctness criteria is described using oracles that can be grouped into types to allow reuse. An example of an oracle is represented in Listing 3.2.

```
/* The oracle type construction defines a class of oracles that can be
    instantiated. */
```

```
oracle type latency_oracle {
/* Oracle types may have instance variables. The max_latency variable, of
     type "period" (a primitive data type representing a time span), contains
      the maximum latency allowed. */
    m_max_latency : period;
/* The oracle's constructor. */
    latency_oracle(max_latency : period) {
        m_max_latency = max_latency;
    }
/* The evaluate method receives as argument the type of computation that is
     evaluated by the oracle and returns true or false depending on whether
     the computation is correct or incorrect. */
    bool evaluate(c : c_type) {
        /* end() and start() are built-in functions that return when a
             computation ended and started, respectively. */
        return end(c) - start(c) < m_max_latency;
    }
}

/* Oracles clauses specify the oracles themselves, instances of their types.
     note that 15s in the argument is a literal value of "period" primitive
     data type. */
oracle limit_15s = new latency_oracle(15s);
```

Listing 3.2: Structure of an oracle

Both oracles and recognizers refer to *computation types*. Computation types have to be declared and may represent architecture-level computations, such as the TKIN, or simple events probed by the system's instrumentation. Conceptually, computation types are similar to Java classes. Listing 3.3 contains the computation type and associated declarations of the probed_ebus_msg computation, which represents a message flowing in the event bus probed by the system's instrumentation.

```
/*All messages in the TKIN. */
enum msg_type {
    mt_best_eqp ,
    mt_eqp,
    // ...
};

/* A struct represents a composite data type, like a Java class. The msg_data
      structure contains information about a lot being processed (the lot's
     unique ID) and the type of message. */
struct msg_data {
    m_lot_id : string;
    m_type : msg_type;
/* Constructor. Works like Java. */
    msg_data(t : msg_type, l : string) {
        m_lot_id = l;
        m_type = t;
    }
/* Function that checks whether two message data refer to the same "thread" (
     lot). */
    bool same_thread(m : msg_data) {
        return m_lot_id == m.m_lot_id;
};

/* A computation type is, syntax-wise, similar to a structure. Computations
     always have a start and end time and are associated with the set of
     architectural elements that contributed to them. These aspects of the
     computations are handle mostly automatically and are, therefore,
     invisible in the syntax. The colon represents inheritance in the object-
     oriented sense. */
computation type probed_ebus_msg : msg_data {
```

```
/* Name of the components that originated the message and to whom the message
      is destined. A single message may have multiple destinations. */
    m_src_name : string;
    m_dst_name : set of string;
/* Constructor. The msg_data constructor is invoked with the colon syntax. */
    probed_ebus_msg(t : msg_type , l : string,
            src_name : string,
            dst_name : set of string)
            : msg_data (t , l) {
        m_src_name = src_name;
        m_dst_name = dst_name;
    }
};
```

Listing 3.3: Event bus message type.

Computation types, recognizers and oracles form the basis of the language used to recognize computations. In general, the diagnosis system's runtime keeps track of which components contributed to each computation as the computations are recognized into higher-level computations. Computations classified by oracles are sent, together with the list of components that contributed to them, to the fault localizer which, in turn, evaluates them using the state-of-the-art Staccato/Barinel toolchain for automatic fault localization [2].

Staccato [1] offers an algorithm that uses a low-cost heuristic for computing a relevant, and valid, set of multiple-fault diagnosis candidates, given a set of computations. The Barinel [3] toolchain is used to compute the probabilities for each diagnosis candidate yielded by Staccato. Barinel assumes critical importance in the performance of our approach for two main reasons:

1) Staccato may yield a long list of diagnosis candidates; and

2) not all diagnosis candidates are equally probable to be the true fault explanation.

When compared to other state-of-the-art approaches to fault localization [2], Barinel yields more information rich diagnostic reports due to the fact that it reasons in terms of multiple faults. We refrain from detailing the approach as it is not the focus of this paper. Further information about the framework and underlying technique can be found in [2].

To perform fault localization, we need to define what the system's architecture is. Our language implements a subset of the Acme language [8] supporting the declaration of component and connect types and their instantiation. Listing 3.4 contains a partial declaration of the system structure.

```
/* A component type specifies a class of components. */
component type proc {
};

/* The ADS is a specialization of the proc component type. */
component type ads : proc {
};

/* An event bus represents a class of connectors. */
connector type ebus {
};

```

```
/* ads_1 is a concrete component. */
component ads_1 = new ads;

/* bus is the event bus. */
connector bus = new ebus;
```

Listing 3.4: Declaration of the system structure.

3.4 Application to the target system

Applying our approach to the system at Samsung Electronics required defining the behavior of the system in terms of the language described in Section 3.3. The main lines of our approach were:

- Each bus message was recognized to produce several messages: a message being *sent* by components and one message (or more, depending on the destinations) being *received* at components.
- The receive/send message pairs that contain call/return behavior, were recognized as a single computation.

  For example, the `BEST_EQP` received at the `ADS` is paired with the `EQP` sent by the `ADS` to produce a `ads_eqp` computation.
- Each pair also produced recognizers that detect behavior failures on timeouts.

  For example, `ads_eqp_no_response` recognizer recognizes a `BEST_EQP` for which *no* `EQP` exists.
- Sequential computations were then bound together in a single computation that represents a sequential flow. We ended up with three sequential flows: `eqp_prep` which corresponds to the flow until the `EES` decides on a pass or fail, a `test_pass` which corresponds to the "pass" optional section in the protocol flow and a `test_fail` which corresponds to the "fail" optional section in the protocol flow.

  For example, the `ads_eqp` is matched to the `mos_start` (recognized from a `BEST_EQP` detected at the `MOS`) to yield a `mos_ads_eqp`. The `mos_ads_eqp` is matched with a `mos_prep` (recognized as a `EQP`/`PREP_EQP` pair at the `MOS`) to yield a `mos_eqp_prep` and so on.
- The alternative flow was modeled using four recognizers: one that recognizes a `eqp_prep` and a `test_pass` into a `eqp_pass`, one that recognizes a `eqp_prep` and a `test_fail` into a `eqp_fail`, one that ensures either `test_pass` or `test_fail`, and one that recognizes a `single_run` from either an `eqp_pass` or an `eqp_fail`.
- Loops were modeled using recursion and loop limitation by placing a counter in the computation.

> For example, the tkin computation type has an attribute m_count that specifies the run count. A tkin can be recognized from another tkin and a single_run. The newly generated tkin has a m_count which is one value higher the the previous tkin.

It is worthy discussing more deeply the matching of call / return patterns within the TKIN protocol. If the contract for the ADS is, as is the case, to *always* reply with an EQP to a BEST_EQP request, then receiving an EQP and not replying with a BEST_EQP is a failure regardless of whether there is any other component involved. However, if the contract for the ADS were to only reply to *some* requests (for example, invalid requests would not have a response), then the pair BEST_EQP/EQP would not be enough to pinpoint a failure in the ADS component and the call / return pattern could not be applied.

The previous discussion hints at two important aspects of our approach: (1) because behavior is formally modeled, it reduces ambiguity in system specifications, and (2) the small pieces of the protocols follow patterns that are reusable across multiple systems, while the large, complete protocols generally do not.

Chapter 4

Metrics for evaluation

To evaluate how well the diagnosis algorithm performs, we need to identify a set of appropriate metrics. There are two main areas of metrics that are applicable in domains sharing similarities with ours: metrics used for evaluation of classifiers and metrics used for evaluation of fault localization algorithms.

Our algorithm has similarities with classifiers in machine learning: it will output a classification of which components are faulty and which are not. In that sense, the classic metrics of precision and recall, or one of its variants, seem to be relevant. However, those statistics are applicable only in binary cases [13] and our classifier outputs a *probabilistically ranked* classification. Many components – maybe even all – may show up as outputs but their *probability*, or *rank* is relevant: if the faulty component has probability of 0.99 and all the others have probability $\frac{0.01}{n-1}$ where n is the total number of components, this is generally seen as a good classification although it has a low value of precision. In the presence of few faulty components, recall will always tend to the high or low spectrum. These metrics are, therefore, not applicable.

The fault localization literature uses ranking for evaluation: the real probability attributed to diagnosis is only relevant in comparison with the others. The rationale behind this metric is that developers will tend, in order to correct the faults, to inspect components by rank order and, therefore, the lower the rank of the faulty component, the more effort will be wasted. This metric is applicable to evaluate run-time diagnosis.

However, at runtime, timing (performance) also becomes an important issue. If at design time, the time taken for diagnosis is not relevant, at runtime it may be critical for autonomic recovery. Also, at runtime, many diagnoses can be produced for the same system and they may yield different values. Therefore, we introduce two metrics:

failure identification time (FIT) that measures the time between when the fault is activated and when the diagnosis system presents some information that *something* is wrong;

diagnosis stabilization time (DST) that measures the time between failure identification and diagnosis stabilization: when the system decides on which diagnosis to consider final.

While for some systems – like the one presented in this case study – it is the sum of both metrics that is relevant, in other circumstances each metric may have different impacts: e.g., if a breach in a high-security system is detected (failure identification time) we may want to disconnect the system from the network immediately even before computing which part of the system was breached (diagnosis stabilization time).

Chapter 5

Evaluation

We evaluated the system by defining the computations using the specification language outlined above. We identified a set of *scenarios* in which the system would fail in some predicted way. These scenarios were designed to replicate the types of real problems encountered at Samsung Electronics.

1. `EES` sends a `FAIL` message and, 3s later, sends the `PASS` message. Because the `MOS` subsystem responds to both events, this starts two parallel flows: one for the successful evaluation, which ends with a `TKIN`, and another for the unsuccessful evaluation, which ends with a retry of the whole process.
2. `TC` database is too slow to respond.
3. `TC` sends `CHECK_EQP` to `ADS` instead of `EES`.
4. Active `MOS` fails and is later replaced by passive `MOS`.
5. `MOS` retries four or more times.
6. `ADS` does not issue `ADB_Q1`.
7. The `ADS` database is too slow to respond.

Evaluation addressed the two main goals stated in the introduction:

1) diagnostic accuracy, and

2) diagnostic performance.

Based on our description in Section 4, we measured the accuracy of the system as the rank of the faulty component in the diagnosis and we measured performance by computing the failure identification time and diagnosis stabilization time.

We also computed another measure, not directly related to the diagnosis system, but of relevance to Samsung Electronics: an estimate of the maximum throughput of the system. Since every recognizer can be run in a different system, the maximum throughput of the system is limited by the amount of time the slowest recognizer takes to perform its evaluation.

Scenario	Component[†]	Rank[‡]
1: EES sends FAIL and PASS	EES	1
2: TC database is too slow	TCDB	1
3: TC will send message to ADS instead of EES	TC	1
4: MOS fails and is replaced by standby	MOS.1	1
5: MOS will retry 4 times	MOS.1*	1
6: ADS will not issue ADS_Q1	-**	-**
7: ADS database is too slow	ADSDB	2***

Table 5.1: Evaluation of performance metrics in the scenarios.

[†] Component where the fault was injected.
[‡] Average rank among 10 scenarios.
* MOS.1 was the active MOS.
** No failure was detected by the diagnosis system in this scenario.
*** ADS and ADSDB were both ranked with an equal probability of 0.5 in all 10 scenarios. We count as 2 as a developer / system maintainer may want to inspect the ADS before the database. We took the most conservative approach.

5.1 Accuracy evaluation

The average result of measuring accuracy in 10 scenarios is presented in Table 5.1. These results show that the system was able to correctly detect a failure in all scenarios except scenario 6. Because databases are a source of performance bottlenecks at Samsung Electronics, it was not possible to determine communication between ADS and ADSDB. Consequently, the diagnosis system has no way of distinguishing between the slowness of ADS and ADSDB; each component is equally likely to be the cause of the slowness. Interestingly, we are able to distinguish the slowness of TCDB. Because the two TC components connect to the same database, slowness in both increases the probability of the fault being localized in the database itself.

5.2 Performance evaluation

Performance metrics evaluated in all scenarios are shown in Table 5.2. For each evaluation scenario we ran the scenario 10 times and collected the two performance metrics discussed in Section 4. The fault localizer in Figure 3.1 outputs the results regularly (once every second).

As noted earlier, the FIT is the time between the fault being activated and a diagnosis being produced that includes an identification of a failure. If this diagnosis result already included the failed component ranked in the first position, then the diagnosis stabilization time (DST) is 0. In several scenarios, a later diagnosis never placed another component in first position, meaning that most scenarios have an instantaneous DST. In scenario 2 however, because `TCDB` was not instrumented, detecting that it was the source of the failure required more data and, therefore, DST is greater than 0.

The minimum FIT column is added for reference. It is the minimum theoretically possible FIT that would detect the failure. In scenario 1, for example, the `FAIL` message is sent 3s after the `PASS` message so, before that, no failure can be detected although the invalid code has already been started. In some examples, like this one, we could have deducted the minimum FIT from the FIT but due to scenarios in which the timing is not so predictable (like introduced database slowness in scenarios 2 and 7) this would not be consistent.

Scenario	avg FIT[†]	std FIT[††]	Min FIT[†††]	avg DST[‡]	std DST [‡‡]
1: `EES` sends `FAIL` and `PASS`	5,506	319	3,000	0	0
2: `TC` database is too slow	13,506	3,212	5,000-15,000	436	432
3: `TC` will send message to `ADS` instead of `EES`	17,708	309	15,000	0	0
4: `MOS` fails and is replaced by standby	15,197	1,505	10,000-15,000	0	0
5: `MOS` will retry 4 times	2,861	0,343	0	0	0
6: `ADS` will not issue `ADS_Q1`	-*	-*	-*	-*	-*
7: `ADS` database is too slow	10,204	1,131	5,000-10,000	0	0

Table 5.2: Evaluation of performance metrics in the scenarios. Results in milliseconds.

[†] Average failure identification time.
[††] Standard deviation of failure identification time.
[†††] Theoretical minimum failure identification time.
[‡] Average diagnosis localization time.
[‡‡] Standard deviation of diagnosis localization time.
[*] No failure was detected by the diagnosis system in this scenario.

Chapter 6

Lessons learned

The application of our diagnosis framework to the system at Samsung Electronics provided us with confirmation that several of our assumptions appear to be supported in an industrial setting and also provided us with insight into some other areas.

Decomposing computations is critical to the success of the diagnosis. As a result of the decomposition, failures can be identified not only in the high-level computations but, sometimes, also in lower level computations. This result allows diagnosis sometimes to be performed in a much smaller set of components yielding a much faster response and a higher accuracy. For example, a lack of `EQP` response allows inferring immediately a fault in the `ADS` component due to the request / response low-level computation model.

Even with strict protocols, there are scenarios where reasoning-based diagnosis analysis is of added values. The previously mentioned decomposition allows local detection and identification of the source of the failure in some cases. This is consistent with industrial practice in which many instances of this reasoning – like heartbeats or timeouts – are used. However, in some cases this is not sufficient. As seen, for example, in scenario 2 where the `TC` database slows down, more complex reasoning may be required to accurately pinpoint the source of the failure.

Behavior specifications can be built up using patterns that are reusable across systems. The computations that comprise the behavior specification we build are defined on top of smaller patterns of well-known computation styles like call/return, alternative flows or loops. Therefore, although different systems may need to specify their own idiosyncratic high-level computations, a significant number of specification blocks may be reused across systems.

Domain experts are able to provide correctness specifications for high-level computations. Domain experts would not be able to explicitly state that within 15 seconds after a `BEST_EQP` message had been seen in the event bus, an `EQP` message from the same ADS to the same MOS with the same `lot_id` field should be seen in the event. This reasoning involves a great deal of low-level detail. However, domain experts *were* able to state that the ADS should respond within 15 seconds of a request for an equipment. This means high-level computations have an abstraction level that matches the domain experts' understanding of the system and are, therefore, a good level to write correctness specifications.

Instrumentation design must consider a trade-off between system performance, diagnostic accuracy and diagnostic performance. The limitation on the observability of some events creates uncertainties which, in some cases like scenario 2 were solved by the reasoning-based analysis with a performance penalty. But in other cases, like scenario 7, this was not possible. However, instrumenting the connection between the ADS and its database was not possible because it was considered by Samsung to introduce an unacceptable performance penalty.

This means that diagnostic accuracy and diagnostic performance have to trade with other system quality attributes. In this system they traded with system performance but in other systems they may have to trade with other quality attributes. For example, probing some connections may lead to easier information leaks, so security could be another quality attribute that would be negatively affected by the introduction of autonomic diagnosis.

Chapter 7

Conclusions and future work

The results indicated by this case study are encouraging as they show that the expected results match practical experimentation. In addition, the study also confirmed that several of our assumptions hold at least in this practical scenario. These conclusions are, however, still bound to be the result of a *simulation* of the real system.

We strongly believe that, in spite of being shown to work on a simulator, our work would be usable in the real industrial setting: adapting the diagnosis system to the real system will require some engineering, but will not raise any new fundamental problems. The diagnosis system can receive the messages from the real system's event bus, just as it receives messages from the simulator itself; the real system events have all the information required for the diagnosis to work; the timings on the simulator are randomized and the real system's timings would also be seen as random, although, most likely, with a different distribution.

The main threat to the validity of the simulation is scalability: our simulated system contains only two instances of the `MOS` and the `TC` and only one instance of the `ADS` and the `EES`. The performance of our recognizers and oracles can be made independent of the number of components of the architecture through horizontal partitioning of the event space as each one is an individual process and they use an event bus to send information. Their number increases but they may be run in parallel if we need to scale up. The point of contention may be the fault recognizer whose complexity raises with the number of components. However, experimentation at design time in [2] has shown that it can handle larger numbers of components than Samsung currently has.

With respect to future work, there are two main complementary paths that we plan to follow: the industrial path and the research path. On the industrial path, we plan to continue to work to bring this system into Samsung Electronics' real system.

On the research path, integration of diagnosis with a self-adaptive framework is a critical piece. It is clear that diagnosis can be used to drive self-repair. But because diagnosis introduces the need to perform trade-offs (for example, between performance and accuracy), it is also reasonable to assume that, ideally, the self-adaptive control loop should tune diagnosis at run time.

Also, in this work we had to manually handle the problem of non-observability: we added the databases to the computations because we knew they were involved even though we did not observe the events. However, we think this approach is more general. The observed behavior could be a projection of the complete behavior in which non-observable computations have been removed. The recognizer and the oracle will have to be updated accordingly, but further research is required to automate this process and understand the implications for diagnostic accuracy and performance.

Acknowledgment

This research was supported by Grant No. CNS 1116848 from the National Science Foundation and by and by a grant from the Foundation for Science and Technology via project CMU-PT/ELE/0030/2009. and by FEDER via the Programa Operacional Factores de Competitividade of QREN (FCOMP-01-0124-FEDER-012983).

Bibliography

[1] R. Abreu and A. J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In V. Bulitko and J. C. Beck, editors, *Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA'09)*, Lake Arrowhead, California, USA, 8 – 10 July 2009. AAAI Press.

[2] R. Abreu and A. J. C. van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation. *Artificial Intelligence*, 174(18):1481–1497, 2010.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Spectrum-based multiple fault localization. In G. Taentzer and M. Heimdahl, editors, *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, Auckland, New Zealand, 16 – 20 November 2009. IEEE Computer Society.

[4] N. Cardoso and R. Abreu. A Kernel Density Estimate-based Approach to Component Goodness Modeling. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI'13)*, 2013 (to appear).

[5] P. Casanova, D. Garlan, B. Schmerl, and R. Abreu. Diagnosing architectural run-time failures. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* To appear., 20-21 May 2013.

[6] P. Casanova, B. R. Schmerl, D. Garlan, and R. Abreu. Architecture-based run-time fault diagnosis. In *Proceedings of the 5th European Conference on Software Architecture (ECSA'11)*, pages 261–277, 2011.

[7] C. Chen, H.-G. Gross, and A. Zaidman. Spectrum-based fault diagnosis for service-oriented software systems. In *Proceedings of the 2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, July 2012.

[8] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[9] W. Humphrey. The software quality profile. *Software Engineering Institute, Carnegy Mellon University*, 1996.

[10] L. Kuhn, B. Price, J. de Kleer, M. Do, and R. Zhou. Pervasive diagnosis: Integration of active diagnosis into production plans. In D. Fox and C. P. Gomes, editors, *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pages 1306–1312, Chicago, Illinois, USA, 13 – 17 July 2008. AAAI Press.

[11] L. M. Ottenstein. Quantitative estimates of debugging requirements. *IEEE Transactions on Software Engineering (TSE)*, 5(5):504–514, 1979.

[12] J. Pietersma and A. van Gemund. Benefits and costs of model-based fault diagnosis for semiconductor manufactoring equipment. In *Proceedings of the 17th International Symposium on Systems Engineering (INCOSE07)*, San Diego, CA, USA, June 2007.

[13] D. M. W. Powers. Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. Technical Report SIE-07-001, School of Informatics and Engineering, Flinders University, Adelaide, Australia, 2007.

[14] RTI. Planning report 02-3: The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, National Institute of Standards and Technology, 2002.

[15] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication.* Univ of Illinois Press, 1949.

[16] J. Tan, X. Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan. Kahuna: Problem diagnosis for mapreduce-based cloud computing environments. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 112–119. IEEE, 2010.