Injecting Inconsistent Values Caused by Interaction Faults for Experimental Dependability Evaluation

Bogdan Tomoyuki Nassu, Kiyonobu Uehara, Takashi Nanya
Research Center for Advanced Science and Technology, The University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo, 153-8904, Japan
{bogdan, uehara, nanya}@hal.rcast.u-tokyo.ac.jp

Abstract

Interaction faults caused by a flawed external system designed by a third party are a major issue faced by interconnected systems. Fault injection is a valuable tool for evaluating the dependability of such scenarios. Several types of errors caused by interaction faults may be injected by existing approaches, even though previous work focused on other types of faults, such as hardware and software faults. This is not the case of inconsistent values — data that is correctly received and syntactically correct, but inconsistent with what it should represent. In this paper, we propose a novel methodology to inject inconsistent values caused by interaction faults, including hand-defined, random and semantically significant values. We also describe a simulation tool which implements the proposed mechanism to aid dependability evaluation in a system that uses the Universal Plug and Play standard to communicate.

1 Introduction

More and more, networked systems built from interconnected sub-systems become widespread. The Internet reach is already worldwide, and consumer electronics companies now aim at devices which connect to each other to form ubiquitous systems [22, 7]. However, as communication protocols become more complex, problems such as misinterpreted specifications or insufficient testing may result in a sub-system being released with an incorrect or incomplete implementation of a communication standard. Dependability evaluation in such scenarios is a major concern.

Dependability evaluation is a highly complex task. To cope with such complexity, several approaches may be used, ranging from the analysis of abstract models to the observation of a system’s behavior under real fault conditions. Fault injection consists of deliberately inserting artificial faults in a target system, so that its behavior under the presence of faults may be experimentally evaluated [21]. This approach can be used, for example, to evaluate the coverage and latency of mechanisms for error detection and recovery; to test fault tolerance architectures; or to analyze the impact of failures in the target system. Fault injection is specially useful in cases where obtaining field data is too difficult or expensive, or when controlled and easily repeatable conditions are needed.

Research on fault injection usually focuses on hardware [15] and software faults [12, 16, 17, 13]. Some work also address communication faults in parallel or distributed systems [3, 8, 9, 14], usually aiming at transmission errors in the transport or lower layers. In this paper, we also consider networked systems, but address the injection of interaction faults — i.e. faults originated from an external system [2]. This is the same type of fault covered by approaches that inject operator faults [4, 19]. However, these approaches consider interaction with a human operator through a user interface, while we consider interaction with a flawed external sub-system using a communication standard.

Most types of errors caused by interaction faults in our scenario can be injected by existing techniques or tools (e.g. omissions, timing errors, etc.). For that reason, we focus on a specific type of error: inconsistent values, data that is correctly received and syntactically correct, but inconsistent with what it should represent. Though existing approaches are able to inject data errors [8, 3, 17, 14], we focus on a different scenario and, more important, on a different level of abstraction, where both the syntax and the semantics (meaning) of the affected data are taken into account.

In this paper, we propose a novel architecture and methodology to inject inconsistent values caused by interaction faults, including hand-defined, random and semantically significant values. We also describe a simulation tool which implements the proposed mechanism considering the Universal Plug and Play [6] standard. The purpose of this tool is to generate communication logs, both correct and containing injected inconsistent values, that can be used for expe-
rimental dependability evaluation.

The remainder of this paper is organized as follows. In section 2, we define the system and fault models, detailing a scenario where interaction faults occur and characterizing inconsistent values. In section 3, we discuss previous work on fault injection in different settings, and their relation with our scenario. In section 4, we describe some real fault cases and propose a new mechanism that can be used to inject inconsistent values. In section 5, we describe a fault injection tool which implements the proposed mechanism. Section 6 concludes the paper and points to future work.

2 System and Fault Models

In this section, the considered system and fault models are described. We detail a scenario where interaction faults occur, and characterize inconsistent values.

2.1 System Model

We consider a networked system built from sub-systems, or units, which use a communication standard to interact. Interaction occurs with the exchange of a sequence of messages, each containing a set of data fields. Each field is identified by a name (“Number of Connections”, “Action Type”, “Price”, etc.), which can be defined by headers, special tags, or the position inside a message, for example.

As interaction faults are originated from external sources [2], the system under observation consists of one correct unit interacting with at least one faulty external unit. No faults are originated from the correct unit. Furthermore, we restrict faults to a single standard. Any protocols below the considered standard are abstracted as a fault-free “communication interface”. Note, however, that faults can propagate to higher layers. Figure 1 illustrates the system model which will be emulated through fault injection.

2.2 Interaction Faults and Inconsistent Values

In our scenario, interaction faults occur when the external unit exhibits a behavior that deviates from the specification of the standard (which is assumed to be correct); or when it sends a message containing incorrect data. We do not consider faults caused by transmission errors in the physical layer — injection for this type of communication fault is already addressed by previous work such as [8, 3, 9, 14]. That means messages are not corrupted or lost in the communication channel, but are indeed generated containing incorrect data or are not sent.

Interaction faults may result in several types of errors outside the external unit. Most error types are well-known, as they can also be caused by other types of faults. These errors include, for example, omissions, incorrect timing and incorrect sequence. These types of errors can be injected by existing tools or techniques, and are not considered in this paper.

In this work, we focus on a specific type of error: inconsistent values. This is a type of data error, which occurs when a message is received inside the expected timing constraints and sequence. Furthermore, the message is correctly formatted, and contains all the required fields with syntactically correct values inside of them. However, one or more fields contain a value which is inconsistent with what it should represent — i.e. it is incorrect regarding the semantics of the field. A message may also be considered inconsistent if it contains correctly formatted fields indicating a state or situation different from reality. Examples of inconsistent values include:

- A field called “Number of Users” contains a valid integer, but the integer does not correspond to the actual number of users in the external unit.
- A field called “Home Page” contains a valid web page address, but the address points to an incorrect page.
- A query request for a variable is issued, and a correctly formatted response is received, but the response does not correspond to the value of the requested variable.
- A correctly formatted error message is received, but the described error did not occur.

An inconsistent value can be indistinguishable from a correct value if only its syntactic properties are tested, making this type of error potentially hard to detect or tolerate. Despite these difficulties (or exactly because of them), inconsistent values are very relevant for dependability evaluation, as they do occur in practice. In section 4.1, some real fault cases that resulted in inconsistent values will be discussed.
3 Related Work

Fault injection has been used in several domains, such as fault-tolerant systems [15, 13], parallel computers [3, 14], distributed systems [8, 9] and databases [19]. Different types of faults have been addressed, including hardware [15], software [12, 16, 17, 13], operator [4, 19] and even communication faults in a more general sense [8, 3, 9, 14]. Even though most existing tools focus on specific domains (e.g. a given operating system or communication protocol), the general ideas used in those tools are the same for most types of errors caused by interaction faults. That includes, for example, omitted messages and timing errors. However, this is not the case of inconsistent values, specially in the level of abstraction proposed in this paper.

Even so, there is a relation between the scenario considered in this paper and previous work on fault injection in other scenarios, specially injection of data errors. For example, approaches to inject communication faults [8, 3, 14] usually include means to affect data inside messages. Similarities can also be observed between our mechanism and approaches for robustness testing [17] and software fault injection [12, 13] which inject parameters with invalid or unusual values into API calls to simulate arbitrary behavior. In all cases, incorrect values are being received through an interface — be it a message-passing protocol or an API. However, these approaches usually work on a lower level than our mechanism, focusing on values represented as sequences of bits or numeric values. Our mechanism provides more complex ways of manipulating strings, dealing with syntactic and semantic properties in a higher level of abstraction.

Another area that relates with our work is the injection of operator faults [4, 19], as these are also cases of interaction faults. Operator faults are defined in terms of the user interface [21], and the mistakes it may allow. In our case, the communication standard is the interface which determines the errors that may occur. However, our work is based on a rule-oriented mechanism, instead of using statistical data from real fault cases or human operators [4].

4 Injecting Inconsistent Values

In this section, a general mechanism that can be used to inject inconsistent values is detailed. Before defining the mechanism used for fault injection, we explore some real fault cases, which can provide some insight on particularities of inconsistent values. Though some of the ideas presented in this paper can be applied to any type of data, we focus on data types that can be represented as strings. The reason is that injected inconsistent values must respect the syntax for their data type: for example, an inconsistent video in MPEG format must be a playable MPEG video. That makes fault injection dependent of the specific data type. However, strings are general enough to represent a multitude of more specific types — e-mail addresses, names, web pages, and even numbers may be represented by strings.

4.1 Real Fault Cases

Representativeness is one of the biggest challenges for fault injection mechanisms in general [21]. This issue can be summarized in one question: is the model for the injected faults able to represent actual faults that may affect the target system? For example, it is generally accepted that bit-flip errors can represent transient hardware faults very well. However, for more abstract or complex fault models, verifying the representativeness is a difficult task. A good solution for this problem is checking how well a large base of real fault cases is represented by the model [12, 16]. Unfortunately, obtaining a large number of real fault cases is also difficult for some scenarios — and in some cases is the very reason why fault injection is necessary.

In this work, we do not validate the fault model by comparing it to field data in a large amount — this remains as an open problem to be addressed in the future. Even so, we have analyzed a number of real fault cases in order to get some insight on particularities of inconsistent values. The analyzed data consists of reports of faults found during the design phase of some devices, including captured communication logs in pcap [1] format. The devices communicate using Universal Plug-and-Play (UPnP) [6, 7] — an architecture for service exchange between electronic devices; based on well-known protocols, such as IP, HTTP and XML, as well as some specialized protocols.

In the fault cases, devices such as a network camera, a printer or a television interact with commercially available UPnP internet routers that (supposedly) follow the specification in [5]. These third-party routers exhibited a faulty behavior when interacting with the other devices, and are the external units in our system model. Figure 2 shows a typical fault case setting.

We have analyzed 14 fault cases which resulted in 17 inconsistent values and 6 omissions — several different errors may occur in a same fault case. The omission cases are not considered in this paper. Below, we summarize the characteristics of the observed inconsistent values (for space reasons, the details of the analysis are not included).

1. In 1 fault case, the external unit has a valid external IP address, but all the requests for a GetExternalIPAddress action receive the value “0.0.0.0” in the response field.

2. In 5 fault cases, an AddPortMapping action is requested, and an error message is received, indicating

---

1 Data gently provided by Matsushita Electric Industrial Co., Ltd.
a conflict between the requested port mapping and an existing mapping, or invalid arguments in the request. Though the error message is correct itself, it indicates a conflict when there is no conflict at all, or invalid arguments event though the request is correct. It can be considered that all the fields indicating the error status are inconsistent with the real state of the external unit or with the issued request.

3. In 4 fault cases, the actions received expected error messages as responses. These error messages were expected because they correctly indicated the actions could not be performed considering the internal state of the external unit. However, the error messages contained error codes and/or descriptions which were not suitable for the requested actions. In one case, the error description actually seems to be only a sequence of meaningless arbitrary characters.

4. In 2 fault cases, a port mapping is requested and successfully added. However, a later request for GetSpecificPortMappingEntry returns an error message or a different mapping than the one which was added — i.e. the response is inconsistent with the real mapping.

5. In 4 fault cases, a field contains a numeric value outside the specified range.

6. In 1 fault case, the messages had meaningless strings in the HTTP CONTENT-TYPE header.

Even though the number of analyzed cases is quite small, some preliminary conclusions can be taken from this analysis:

- First and foremost, inconsistent values seem to be rather common in practice, at least in this particular system instance (considering UPnP routers).
- Several inconsistent values do not lead to failures in the system — this is specially true for some descriptive fields that are not used by the correct unit and for some out-of-range values that are used without testing.
- All the errors are repeated if a specific external unit receives the same sequence of messages. In some cases, different correct units try to perform the same actions, and receive the same inconsistent values — i.e. an error is repeated if a certain condition or context is repeated. That observation leads us to the conclusion that several interaction faults may occur due to design faults in the external unit. These faults make the external unit fail in operation time, becoming the cause of interaction faults in the top-level system.

4.2 Fault Injector Architecture

As said in section 4.1, interaction faults in a system may be caused by design faults in one of its sub-systems. For that reason, a possible approach for injecting inconsistent values would be actually injecting software faults in the external unit. This could be done by modifying the external unit’s executable code, like done in [16]; or it’s source code, including mistakes such as non-initialized or non-updated variables, incorrectly initialized constants or data files, and flawed algorithms. However, listing all the design faults that may result in inconsistent values is a difficult task, specially if we want to have control over the resulting values and when they appear. Moreover, in a real development environment, the external unit is sometimes a “black box”, designed by a third party. For those reasons, we propose injecting faults by modifying the messages sent by the external unit, instead of the external unit itself.

By injecting inconsistent values directly into the messages, we actually inject errors, even though the process is still called “fault injection”. This type of “data injection”, which changes messages and data after they have been correctly generated, can also be seen in existing work [12, 3, 8, 17], and is justified by the fact that different faults may result in errors of the same type — thus, it is possible to emulate faults by injecting the errors they cause.

Though this approach is more straightforward than changing the external unit’s software, it still has some problems which demand attention. One problem is that the injected errors should not affect protocols in layers below the considered standard. If the message is split and encapsulated, the injector would have to re-assemble it, modify the contents, and split it again, recalculating error detection codes such as CRC or checksums. Moreover, this process must be
fast enough so that the injection of inconsistent data does not actually cause timing errors — intrusiveness must be kept to a minimum. These problems may be minimized if the fault injector acts inside one of the units, modifying the message before it leaves the external unit or after it is already re-assembled in the correct unit. As the external unit is sometimes a “black box”; as said above, the second option is usually the best.

Figure 3 shows a general view of the fault injection environment. The external unit sends a correct message through the communication interface (the protocols below the considered standard). After the message is correctly transmitted by the lower-level communication interface, the fault injector acts over the message and generates a new message containing inconsistent values. The inconsistent message is then received by the correct unit. The fault injector generates a fault log and is controlled by a set of fault rules, which define when faults occur and how they affect messages. Fault rules are further detailed in the next subsection.

Each fault rule defines how a correct string $s$, inside a given field $F$, can be transformed into an inconsistent value $t$. To this purpose, a fault rule is divided into two parts [21]:

- A fault trigger describes the conditions that should be met in order to inject the fault — i.e. which specific instances of field $F$ are affected by the rule.
- A fault type describes the actual change in the target system state induced by the fault — i.e. which rules of transformation are applied to the correct string $s$ in $F$ so that it is turned into an inconsistent value $t$.

Below, we show how these concepts are used in our fault injection architecture.

4.3.1 Fault Triggers

A fault trigger defines the circumstances where an inconsistent value is injected. For example, a rule may be triggered only when a given task is being performed, or when a given field is present. In our architecture, the information used to define a fault trigger is:

- **Affected Task**: Specifies the type of message to be affected — e.g. a certain action, responses for requests of a given type, etc. The types of messages are defined by the fault injector, based on the protocol specification: when the fault injection tool is implemented, the messages that may be sent by the considered standard are divided so that each type of message belongs to one, and only one, task. Note that the definition of the boundaries for a task is subjective: a single standard may be divided in several different ways, with a level of granularity ranging from very low — the entire protocol is a single task — to very high — each message defines a different task. Tasks can also be divided in sub-tasks, forming an hierarchy. If a top-level task is specified by the fault rule, all the sub-tasks are affected. If no task is specified, the fault rule affects all types of messages.

- **Affected Field**: Identifies the field $F$ that will have its contents changed. The field names are defined based on the protocol specification. The same way as with tasks, fields may be divided in hierarchies, and several sub-fields may be affected if a top-level field is specified. If the specified field does not exist in a message, the rule is not triggered. If no field is specified, the entire message is replaced.

- **Conditions**: The fault is only injected if certain conditions hold true. All the conditions must be met for a fault rule to be triggered. Assuming that identifiers $id1$ and $id2$ are values or field names, we define the
following conditions (other conditions may be defined for more specific protocols or data types):

- \(id1 = id2\) and \(id1 \neq id2\)
- \(id1 > id2\) and \(id1 \geq id2\): can be used only with strings representing numbers.
- \(\exists id1\): true if \(id1\) is the name of a field present in the message.
- \(\text{delay}(id1)\): the fault is triggered only once, after \(id1\) cases where all the other conditions are met.

A fault trigger defined as above may be used to simulate several circumstances for the occurrence of a fault. However, this approach depends on context-related information, thus requiring each message to be interpreted by the fault injector. Depending on the communication standard, this process may be simple (e.g. the context-related information can be retrieved from headers) or complex (e.g. the information may only be obtained by tracking all the communication flow). This should be taken into account when implementing the fault injector.

### 4.3.2 Fault Types

The second part of a fault rule is the fault type. The fault type determines, once the fault is triggered, how the correct communication flow (i.e., the semantics of the communication) is turned into the inconsistent value \(t\). Common techniques for changing data values include random bit inversions; fuzzying (generating random values, ignoring the correct data); applying rules such as “increment” or “decrement” for numeric values; or simply setting \(t\) to a predefined value. In [17], it is shown that even though all these techniques may generate any possible value for a given field, the distribution of the errors is very different depending on how the values are generated. Although that work focused on a scenario which is very different than ours, the same general conclusion can be extended for our case.

To obtain a better distribution in the error space, our fault injection architecture combines hand-defined and randomly generated values. Furthermore, we add means for the generation of values that still respect the syntax rules for their data type. These errors cannot be detected by syntax rules, and should be taken into account when error detection mechanisms are evaluated. Finally, we define a new concept that can be used to generate semantically relevant inconsistent values: past occurrence classes.

Past occurrence classes are based on the observation that the values received in a field \(F\) during a correct communication implicitly carry some information about the meaning of \(F\). For example, a field called “city” may receive such values as “Tokyo”, “New York” or “London”. Implicitly, there is the knowledge that these are values suitable for the “city” field in a correct communication, even though the fields and values do not have their semantics analyzed. Semantically relevant inconsistent values are a useful addition to our architecture, because this type of error may be more difficult to detect than meaningless strings, depending on the error detection mechanisms. The possible past occurrence classes for an inconsistent value \(t\) compared to the correct value \(s\) are:

1. Unknown string (ex: for \(F = \text{“name”}\), \(t = \text{“Bogd”}\))
2. Known string never seen in \(F\) or any other field (ex: for \(F = \text{“name”}\), \(t = \text{“house”}\))
3. String seen in other field but not in \(F\) (ex: for \(F = \text{“name”}\), \(t = \text{“Tokyo”}\))
4. String seen in \(F\) before (ex: for \(F = \text{“name”}\), \(t = \text{“Bogdan”}\))

Note this classification is history-dependent, i.e. the past occurrence classes can’t be defined only by comparing the inconsistent with the correct value, they are defined over a given history of past communications. This history may be given, for example, as a database containing specially chosen values, or values observed during a learning stage of correct communications.

Each fault type is defined by the following information:

- **Pattern to Replace**: Given as a regular expression, specifies the parts of the original string that must be replaced by a different string. This pattern may be used so that the syntax of specific data types are still respected by the inconsistent value. For example, suppose \(s\) is the e-mail address “foo.bar@foobar.co.jp” and assume regular expressions in the Java format [11]:

  - The pattern “@\w+\.” replaces the substrings between the “@” symbol and a dot (“@fooobar.”).
  - The pattern “\w+\z” replaces only the last term after a dot (“.jp”).
  - The pattern “.+@” replaces only what comes before the “@” symbol (“foo.bar@”).

If the specified pattern is not found, \(s\) remains unchanged. If no pattern is specified, the entire contents of the field are replaced. If the entire message is being replaced (i.e. if no field was specified to be replaced), the pattern is ignored.

- **Replacement String**: Defines the string that will replace the strings matching the specified pattern. The replacement string may contain escape sequences, used for generating random inconsistent values. Escape sequences are ignored if the entire message is being replaced (i.e. no field to replace is specified). Otherwise, the “\” character is ignored unless it defines one of the following escape sequences:
– “\"” is the “\"” character.
– “\[l” generates a random string containing letters.
– “\[d” generates a random string containing digits.
– “\[w” generates a random string containing letters or digits.
– “\[r” generates a random string containing specific characters (see below).
– “\[p” chooses a random string based on a given past occurrence class (see below).

If no replacement string is specified, the substrings matching the specified pattern are removed from s, or t is an empty string, if no pattern was specified.

- Random String Size: Defines the size for the strings generated by the escape sequences \[l, \[d, \[w and \[r. If no size is specified, the random strings have also a random size.
- Random Strings Characters: Specifies the characters used by the \[r escape sequence.
- Past Occurrence Class: Defines the past occurrence class for strings chosen by the \[p escape sequence.

4.4 Generating Inconsistent Values

We show in this subsection how our mechanism can be used to generate inconsistent values such as the ones described in section 4.1. Note they are just a small subset of the types of errors that may be generated with our mechanism. The numbers below refer to the number used in that subsection.

1. The fault trigger specifies a task (UPnP Control, Action Response), a condition to affect only responses for the GetExternalIPAddress action (exists(“m:GetExternalIPAddressResponse”)), and the field “NewExternalIPAddress”. The fault type specifies only the replacement string “0.0.0.0”.

2. The fault trigger specifies a task (UPnP Control, Action Response), a condition to affect only responses for the AddPortMapping action (exists(“u:AddPortMappingResponse”)), and no field to replace. The fault type has a replacement string, an error message that will entirely replace the correct response.

3. The fault trigger specifies a task (UPnP Control, Action Response) and the fields “errorCode” and “errorDescription”. The fault type can have constant or random strings that will replace the contents of these fields.

4. Use the same techniques as in items 1 and 2 above.

5. Use the same techniques as in item 1 above.

6. Use the same techniques as in item 1 above.

5 Implementing an Inconsistent Value Injector for UPnP

The general mechanism proposed in section 4 was implemented considering the Universal Plug-and-Play [6] standard. This implementation was used to generate inconsistent values similar in nature to the ones found in the fault cases described in section 4.1. Instead of injecting faults in messages from an actual UPnP device (certainly, a viable option), the fault injector was designed to run over a simulation tool we developed. The main purpose of this tool is generating communication logs — both correct and containing injected faults — for dependability evaluation in our scenario.

Simulators have some advantages and some drawbacks if compared with more realistic implementations. Among the advantages, we emphasize the increased control over the experiments, allowing for more flexibility; as well as isolating the considered standard (UPnP) from interference by other protocols. The greatest disadvantage of a simulator is that its accuracy depends on how the simulated environment is modeled. In order to increase accuracy, we only simulate the timing, the high-level behavior of the UPnP devices, and the lower-level communication interface. For the affected standard, UPnP, we used a modified version of an existing open-source UPnP implementation [18]. As errors are being injected in that level, though no messages are actually sent through a communication interface, the simulation is very accurate — all messages are generated and interpreted exactly the same way as in a real implementation.

Figure 4 shows a conceptual view of our UPnP Communication Simulator. Below, these concepts are explained in more detail. For space reasons, it is not possible to detail every aspect of this implementation — only the most important aspects are emphasized. The fault injector was implemented in the Java language, and may be obtained in [20].

- UPnP Device, UPnP Control Point and UPnP Device Specification: UPnP units are divided in two types: devices and control points. Devices offer services, respond to action and query requests, and send messages when they enter the network or when specific events occur. Control points are able to search for devices, issue requests, and receive responses, event messages, or announcements from devices.
Two UPnP devices are simulated, as well as their respective control points. The first device is a simple test device: it has a boolean state variable and can receive requests to toggle it between true and false. The second device is a hypothetical content server device, which we specified for this tool. The content server works as a file repository with user access control — files can be stored in directories, copied, have their properties changed, etc. The high-level behavior of this device is simulated: no files are actually sent or received, but meta-data about them is kept (e.g. size, owner, path, etc.). The complete device specification can be obtained in [20].

- **Event-based Scheduler**: The simulator is based on scheduled events that take place in determined moments. This process is completely event-based — the time counter is not updated in real time, but is increased according to the scheduled events. That means the fault injector intrusiveness is not an issue, as no matter how long it takes to inject errors, the time counter is always updated as scheduled. On the other hand, several timing characteristics are hard to simulate; but this is not a major drawback, as our work focus on data errors, and not timing errors.

- **Experiment Engine and Experiment File**: An experiment file describes the environment for a series of communications between the device and control point. It contains an XML structure listing the timestamps and and the events that occur, as well as their parameters. The experiment engine is a coordinator which reads the information in an experiment file and uses it to set up the other components in the simulator.

- **Communication Interface and Communication Log**: UPnP runs over some protocols such as UDP, IP and TCP. These protocols are not simulated at all, and messages are internally sent between devices and control points by the means of simple pointer operations. All the messages are stored in a communication log along with their respective timestamps. The communication log is the main program output.

- **Fault Injector, Faults File and Fault Log**: These components are an implementation of the mechanism described in section 4. Faults can be injected in messages sent by both the device and the control point: both can be the external unit in our model, depending on the experiment. Fault rules are described in XML structures read from files.

  The UPnP tasks were divided as shown in figure 5 — this division is based on the UPnP specification [6], but is only one of the possible divisions. To simplify the process of deciding the type of each message, the simulator passes along with each message a special tag, containing the name of the task the message belongs to. Instead of interpreting the message, the fault injector uses these tags to decide the type of each message.

A typical simulation occurs as follows: first, experiment files and faults files are written using tools provided along with the simulator. These files are then loaded by the experiment engine, which sets up the environment and schedules events. The initial state of the devices is always the same, containing information about several files, specially data about songs obtained from [10]. The simulation begins, and events occur in order, with messages being generated by the UPnP implementation. All the messages are saved in a communication log, while an error log is kept for further use. The simulator may run with or without injecting faults.

  The simulator may also run in a special mode, without injecting faults and saving all the values received in each field. These values are then kept in dictionaries, which are, together with a general dictionary, used by the fault injector to decide the past occurrence classes when the \p escape sequence appears in a replacement string.

  To test the simulator, we have generated several experiment files, containing random actions, query requests and eventing messages. That workload was used to generate dictionaries for past occurrence classes, and to check our fault injection mechanism. We were able to follow the guidelines from section 4.4 and generate inconsistent values such as the ones described in section 4.1, as well as several other types of errors.
6 Conclusions and Future Work

In this paper, we have addressed the problem of injecting inconsistent values caused by interaction faults to emulate a scenario where a correct unit communicates with a faulty external unit. System and fault models were defined, and a novel architecture for fault injection was proposed. The architecture can be used to inject hand-defined, random and semantically significant inconsistent values in messages. Though neither the models nor the mechanism were validated by analysis of field data in a large amount, we have shown how the mechanism can be used to generate inconsistent values found in a number of real fault cases. It must be noted, however, that these fault cases are only a small subset of the types of faults that can be emulated by our approach. We have also described a simulation tool that explores the proposed mechanism to generate communication logs containing injected faults.

Future work includes analyzing a large body of field data to better validate our system and fault models, as well as our fault injection approach. Employing our fault injection tool to experimentally validate error detection mechanisms is also planned as a next step.

References


