Model-based runtime analysis of distributed reactive systems

Andreas Bauer Martin Leucker Christian Schallhart
Institut für Informatik, Technische Universität München
{baueran, leucker, schallha}@informatik.tu-muenchen.de

Abstract

Reactive distributed systems have pervaded everyday life and objects, but often lack measures to ensure adequate behaviour in the presence of unforeseen events or even errors at runtime. As interactions and dependencies within distributed systems increase, the problem of detecting failures which depend on the exact situation and environment conditions they occur in grows. As a result, not only the detection of failures is increasingly difficult, but also the differentiation between the symptoms of a fault, and the actual fault itself, i. e., the cause of a problem.

In this paper, we present a novel and efficient approach for analysing reactive distributed systems at runtime, in that we provide a framework for detecting failures as well as identifying their causes. Our approach is based upon monitoring safety-properties, specified in the linear time temporal logic LTL (respectively, TLTL) to automatically generate monitor components which detect violations of these properties. Based on the results of the monitors, a dedicated diagnosis is then performed in order to identify explanations for the misbehaviour of a system. These may be used to store detailed log files, or to trigger recovery measures. Our framework is built modular, layered, and uses merely a minimal communication overhead—especially when compared to other, similar approaches. Further, we sketch first experimental results from our implementations, and describe how it can be used to build a variety of distributed systems using our techniques.

1. Introduction

Reactive real-time systems are increasingly embedded and, due to modern communication and fault-tolerant bus technologies, also increasingly laid out as *distributed* systems. Often they control safety-critical applications and have already pervaded everyday life, e. g., in terms of automotive control-systems used in present-day cars, mobile phones, or modern aircraft systems.

In general terms, a *real-time system* is one in which the temporal aspects are part of its specification. As such not only the correctness of a computed result is crucial, but also the time at which it is produced. In case of an embedded system, it is usually the environment which imposes a strict frequency upon the system which needs to react and respond, i. e., follow *hard deadlines*. Such systems are more precisely referred to as *reactive systems* [11]. However, not only embedded systems can be reactive; many business information systems are also typically labelled as being real-time sensitive, or reactive. Unlike in the embedded world, however, many deadlines in business information systems are *soft deadlines*, i. e., some of them may be missed by the system without fatal consequences on the environment or even human life.

The design and development of embedded systems, especially in a safety-critical setting such as automotive, for instance, can be guided by the use of *formal methods* [28], such as model checking or deductive reasoning, in order to increase our confidence in the correctness of the system. However, formal methods employed in the design and development process alone cannot guarantee that systems are sufficiently prepared to deal with unforeseen events or even errors, probably induced by the environment. More so, certain assumptions made during the development process, e. g., predetermined fault models, may prove to be inadequate in a real-world setting.

1.1. Related work

Although a lot of today's systems are equipped with custom built-in diagnostic mechanisms, they usually provide insufficient means to distinguish between the *symptoms* of a fault, i. e., an observed failure, and the actual fault itself, i. e., its *cause*. Diagnostics is then often reduced to a mere recording of symptoms. To address this problem, various improvements were suggested as well as implemented, for instance, adding additional knowledge about the system under scrutiny in terms of cause and symptom "tables", reflecting the effects of certain failures [17, 14]. These may be obtained prior from a dedicated hazard and risk analysis, or directly from the engineers who designed the system and know about its possible ways of failure [27, 26, 10]. The downside of these solutions, however, is that such knowledge basically constitutes assumptions, and as such these may be invalidated by the real-world, e. g., when situations occur that are not explicable using this knowledge.

Another approach to obtain a more holistic view on distributed systems has been introduced in [13]: global system properties are monitored using watchdogs, which are transparently distributed amongst the system's components in order to detect violations of these properties. The holistic view is then obtained by exchanging diagnosis messages between the watchdogs, each attached with a time stamp, in order to identify those system parts/watchdogs where an error initially occurred. However, depending on the property, the price to pay for this solution are $O(\binom{n}{2})$ extra messages, which need to be continuously communicated, where n is the number of watchdogs used.

1.2. Contribution

In this paper we introduce a combined framework for a dedicated runtime analysis which avoids many of the problems that currently exist in monitoring and diagnosing distributed reactive systems. We sketch the theoretical foundations for our framework, and provide experimental results from our implementations.

Basically, the framework as is combines two novel approaches, first for detecting failures at runtime, and then secondly for analysing their causes requiring only a minimal communication overhead; in fact, only linear with respect to the number of used watchdogs, and only in case of a system error. Unlike failure detection by means of system monitoring, the identification of failures is only performed using a dedicated system's diagnosis if, prior, a monitor has noticed a certain misbehaviour. As such, there exists no continuous computation and communication penalty for diagnosis, in case the system under scrutiny works as expected.

In contrast to many similar approaches, e. g., [6, 13, 25], we also provide means to explicitly specify and automatically reason about real-time properties and systems, which is an important prerequisite when dealing with hard deadlines. The experimental results demonstrate the feasibility of our approach and hint to the scalability of the methods. Moreover, the framework can be downloaded (see `http://runtime.in.tum.de/`), and is developed and publicly available in terms of an open-source project.

Notice in the remainder of this paper, we refer to our work in terms of a *runtime reflection framework*, indicating a system's ability to reason and reflect about its own operating modes and overall system state at runtime by employing our framework, in a flexible and highly customisable fashion.

Outline. After a brief overview on the overall architecture in the next section, we provide more details on our employed methods for performing runtime analysis. We, therefore, first discuss the background of runtime verification (see Sec. 3), and then of model-based diagnosis (see Sec. 4). Afterwards, we develop, at the end of each respective section, our particular approach and realisation of that according method. Then, in Sec. 5, we provide some technical insights into the implementation of our methods and, finally, conclude the paper in Sec. 6.

2. Architectural overview

In this section we first present an architectural overview on our runtime reflection framework. First, we consider its structure merely in terms of the existing layers, and without regarding in particular the distribution of its underlying components within the layers. Then, we describe the organisation of the individual components of our framework by means of giving a brief intuitive example, reflecting more on the distributed nature of our architecture and the application to be analysed at runtime.

2.1. The layered view

The architecture is a layered and modular one, in that it well-supports a separation of concerns; that is, the different tasks of the analysis are handled by separate layers which communicate only through minimal interfaces, as is indicated in Fig. 1.

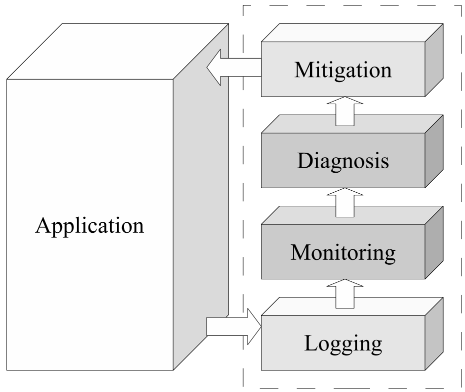

Figure 1. An application and the layers of the runtime reflection framework.

Let the application under scrutiny be a (possibly) distributed reactive system, instrumented and/or annotated to produce an outside-visible stream of (internal) system events.

Logging—Recording of system events. A dedicated *logging layer* in our architecture is then the only part of the runtime reflection framework directly known to the application itself. The distributed application, embedded into the framework, employs special code annotations in order to produce the visible system events, which are then collected and communicated further by our logging layer. The annotations are the only prerequisites, necessary within the application's code, in order to be able to use our framework in conjunction with an application.

Further, the logging layer allows to register so-called *loggers* for observing the stream of system events, and thus, to reflect upon the runtime behaviour of the executed application. A logger might be part of the application itself, e. g., to extract more general statistics on the overall system utilisation, or to record system events merely to a file during a unit-test session. However, when we employ the logging layer in conjunction with the complete runtime reflection framework, we use the layer to deliberately decouple the application's code from the remaining layers in the framework.

In particular, the application's code does not contain any knowledge on the properties which are monitored, and which are then used subsequently for deducing a diagnosis in case of an error. Therefore, we can change the monitored properties and the system description (as used by the diagnosis) even on-the-fly, during their execution without interrupting the running application.

Monitoring—Failure detection. The *monitoring layer* consists of a number of monitors (complying to the logger interface of the logging layer) which observe the stream of system events provided by the logging layer. Its task is to detect the presence of failures in the system without actually affecting its behaviour. It is implemented via *automatically generated monitors* which—each locally with respect to a certain subsystem or system's component—monitor *safety properties* (see Sec. 3).

Intuitively a safety property asserts that "nothing bad happens". Therefore, safety properties impose minimal requirements upon the system which must hold in order to have some sort of a well-defined behaviour. They do not, however, impose a specific behaviour on the system as such. A typical example is the exclusion of certain critical system states, e. g., one always wants to ensure that $\neg(critical_1 \wedge critical_2)$ holds.

If a violation of a safety property is detected in some part of the system, the generated monitors will respond with an alarm signal for subsequent diagnosis.

Diagnosis—Failure identification. We deliberately separate the identification of causes from the detection of failures in terms of a dedicated diagnosis system. The *diagnosis layer* collects the verdicts of the distributed monitors and deduces an explanation for the current system state.

For this purpose, the diagnosis layer infers a minimal set of system components, which must be assumed faulty in order to explain the currently observed system state. The procedure is solely based upon the results of the monitors, and as such, the diagnostic layer is not directly communicating with the application, but rather creates with each diagnosis a "snapshot" of the system at a given time. This bears a major advantage in that no extra messages need to be exchanged between all the monitors in order to obtain a holistic system view.

Our diagnostic layer then infers a system model which incorporates and reflects the observed failures, and compares it with an internal reference model. The differences found constitute possible causes for failure. Basically, this approach is based upon an efficient realisation of the theory of consistency-based diagnosis (see Sec. 4).

Mitigation—Failure isolation. The results of the system's diagnosis are then used in order to *isolate* the failure, if possible. However, depending on the diagnosis and the occurred failure, it may not always be possible to re-establish a determined system behaviour. Hence, in some situations, e. g., occurrence of fatal errors, a recovery system may merely be able to store detailed diagnosis information for off-line treatment.

In the following sections, for brevity, we therefore focus on the first two layers, monitoring and diagnosis, and establish the theoretical foundations for our framework, and sketch its implementation along with some preliminary results.

2.2. The distributed-system view

So far, we merely discussed the tier-structure of our architecture, while we did mostly ignore the distributed nature of it. However, the distribution is oriented towards the layering of the framework: the logging layer and the monitoring layer consist both of a number of different software components, which are distributed throughout the system under scrutiny; that is, depending on the granularity and number of the system's components. Each local monitor then computes a verdict on the locally observed event stream and provides this verdict for further, subsequent diagnosis regarding the system's general status. The diagnosis and mitigation layers, in contrast to logging and monitoring, are realised in terms of centralised components, which collect the information of the monitors in order to compute and react upon a global system view.

For instance, consider Fig. 2, where we show an example application consisting of four distributed components, $C_1, \ldots, C_4$. To monitor the overall system behaviour of

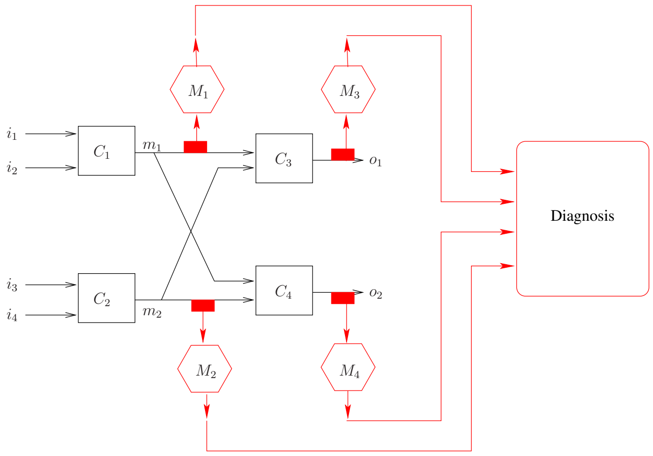

Figure 2. Distributed monitoring & diagnosis.

this application, we employ four dedicated monitoring applications, $M_1, \ldots, M_4$, to the system. Each monitor M_i is then locally observing the output of a single component, C_i, and computes its verdict on the correctness of the observed output stream so far. These distributed verdicts are then transmitted back to the central diagnosis component for further treatment via the application's communication infrastructure which, depending on the nature of the system, may be a physical bus system or merely remote procedure calls, for instance.

3. Runtime verification

Basically, the monitors used for failure detection in our setting are automatically generated from specifications formulated in *linear time temporal logic* (LTL) [20].

In a model-based development process of safety-critical systems (cf. [4]), formal requirements are often formulated in LTL. Then *model checking* [5] can be used to decide automatically, whether the *model* satisfies the property at hand [21].

When implementing the model in terms of software or hardware, discrepancies between the model and the actual system (or the environment for that matter) might come into play. Thus, in order to improve the overall result, one can select the most important requirements to be monitored at runtime, such that crucial aberrations are detected and dealt with accordingly.

In model checking, a complete model of the system is given and all possible *infinite* traces are considered for checking the LTL property in question. In runtime verification, however, we can just examine a *finite* part of a possibly infinite behaviour—the sequence of actions carried out by the underlying system so far. It is therefore important to come up with an adequate semantics for LTL on *finite traces* that extends soundly to the infinite trace semantics. As we argue below, we achieve this goal using a 3-valued semantics for LTL on finite traces.

Besides plain LTL, which is suitable for synchronous systems with a fixed notion of steps, we are often faced with hard real-time constraints in software or hardware systems, especially in the automotive or telecommunication domain. We therefore extend the setting towards a timed version of LTL, namely TLTL, that allows to formulate real-time constraints on the actions observed.

For both logics, we can easily obtain *monitors* that signal the semantics corresponding to the observations so far [2]. The results then constitute the basis for the diagnosis as described in the next section.

3.1. Background: linear time temporal logic

The set of LTL formulae is inductively defined by the following grammar, where AP is a finite set of atomic propositions:

$$\varphi ::= \mathit{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\, U\, \varphi \mid X\varphi \quad (p \in \mathrm{AP}),$$

A LTL formula φ is interpreted over an infinite trace $w = a_0 a_1 \ldots$, where each a_i is actually a set of propositions, identifying the observations of the underlying system. The formula p expresses that in the current instant of the observed trace (a_0), p has occurred. true and the boolean combinations are as expected. An "until" formula, e. g., $\varphi U \psi$, states that ψ holds at a present or some future instant, and that φ holds until then. A "next" formula, e. g., $X\varphi$, states that φ holds in the next time instant.

While the grammar above is complete to define the set of LTL formulas, it is typically—as in our tool set—enhanced by further operators that make specifications more concise, thus, more readable as well as the overall approach more useful in practice. For example, we use a "globally" operator (G) as in $G\varphi$ to express that φ holds now and will hold at all future instants, and a "finally" operator (F) as in $F\varphi$ to say that φ holds at present or will hold at some future instant.

In order to get an intuitive access to LTL specifications, let us briefly get back to our example property from Sec. 2.1, which would be correctly expressed as $G\neg(\mathit{critical}_1 \wedge \mathit{critical}_2)$, where $\{\mathit{critical}_1, \mathit{critical}_2\} \in \mathrm{AP}$. Thus, it says that never both $\mathit{critical}_1$ and $\mathit{critical}_2$ occur at the same time. On the other hand, if we think of a concrete target, such as an automobile, for instance, we may want to make sure that, while the vehicle is running, the key is not removed from the ignition; that is, we monitor the property defined by $G(\neg(\mathit{speed} = 0) \rightarrow \neg(\mathit{ignition} = \mathit{keyout}))$. Here, $(\mathit{speed} = 0)$ and $(\mathit{ignition} = \mathit{keyout})$ are atomic propositions, and $\rightarrow$ denotes logical implication.

3.2. A three-valued approach

Unlike model checking, runtime verification is a *dynamic* method, applicable to white, gray or black-box systems alike. In a nutshell, it works as follows. A correctness property φ, formulated in (some variant) of LTL, is given and an according monitor $\mathcal{A}_\varphi$ automatically generated. The system under scrutiny as well as the generated monitor are then executed in parallel, such that the monitor observes a system component's stream of actions. System actions which violate property φ are then detected by the monitor and an according alarm signal is raised.

However, since a monitor can have at most a *finite* view on the system's behaviour over time, whereas LTL is originally defined over infinite behavioural traces, a semantics for LTL on finite traces has to be defined—but one that goes along with the engineer's expectation that is based on the infinite trace semantics!

Typically, a two-valued (*true*/*false*) semantics on finite traces has been defined and used in runtime verification tools, such as, e. g., [7] or [12]. However, in our opinion, any two-valued semantics is unsatisfactory. For instance, what should be the interpretation of Xp in the last observation of some finite trace? Since the next state has not been observed yet, we do not know whether p holds there. Assigning *false* would make a monitor raise complaints, although no violation has been observed. Assigning *true*, on the other hand, is misleading, since it is not clear whether p holds in the next observation.

On the other hand, consider the formula $\neg p\, U\, init$ stating that nothing bad (p), should happen before the init-function is called. If, indeed, the init-function has been called and no p has been observed before, the formula is *true*—regardless as to what will happen in the future.

In our framework, we have solved this problem by interpreting LTL using a *3-valued semantics*, i. e., with the values *true*, *false*, and ?, where the latter denotes an *inconclusive* verdict, indicating that the behaviour observed so far does not allow to decide whether φ holds or whether it will be violated in the future.

Formally, we define our 3-valued semantics over the set of truth values $\mathbb{B}_3 = \{\bot, ?, \top\}$ as follows. Let $u \in \Sigma^*$ denote a finite behavioural trace. The *truth value* of a formula φ w. r. t. u, denoted by $[u \models \varphi]$, is an element of $\mathbb{B}_3$ and defined as follows:

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases}$$

where Σ^ω denotes the set of infinite behavioural traces and $w \models \varphi$ denotes the standard (two-valued) satisfaction relation of LTL on infinite words, defined for example in [18].

Intuitively, the definition states that a formula only evaluates to $\top$ if, based on the finite prefix observed so far, it is currently *true*, and if there exists no continuation, σ, which may invalidate it; vice versa, for $\bot$. If neither conclusion can be drawn, the truth value of a formula is ?, i. e., inconclusive.

For verification, it is important to know whether some property is indeed *true*, or whether the current observation is just inconclusive. When monitoring a property φ and the monitor signals *true*, the monitor can be stopped, since it cannot report any violation any more. The underlying property of such a monitor requested to watch over the system up-to some moment that has occurred, like in the until example above.

In [2], we have developed an efficient automata-based monitor procedure for our 3-valued logic, abbreviated as LTL_3. Basically, it builds on the well-known translation of LTL to Büchi automata, but substitutes the acceptance condition in that it yields a finite Moore machine for a formula $\varphi \in \text{LTL}_3$ that outputs three symbols, based on the internal state the machine is currently in. The automata are subsequently used to generate code for the actual runtime monitors. Some implementation details are available in Sec. 5.

3.3. Extension towards real-time

Additionally, we have raised our 3-valued runtime verification approach to explicitly deal with timed behaviour in order to be able to monitor real-time properties of reactive systems. To formulate such real-time requirements, we employ timed LTL (TLTL for short), a logic originally introduced in [22], but in the form presented in [23].

The language expressible by a TLTL formula can be defined by *event-clock automata* [1], a subclass of *timed automata*. It was shown in [8] that TLTL corresponds exactly to the class of languages definable in first-order logic interpreted over timed words. Recall that LTL corresponds to the class of languages definable in first-order logic interpreted over (non-timed) words [15]. Thus, it can be considered as the natural counterpart of LTL for the timed setting.

LTL is suited for synchronous systems, where a notion of *step* exists. In each step, the propositions in question (AP) are either *true* or *false*, and a log event read by the monitor is a vector denoting the corresponding truth values.

In the real-time setting, we assume an event-driven architecture. The monitor reads subsequently (notifications of) *events* together with the time when the events occurred. Correspondingly, the atomic entities in our logic are no longer atomic propositions but timed events.

Formally, the syntax of TLTL is defined as follows:

$$\begin{aligned} \varphi \quad ::= \quad & true \mid a \mid \lhd_a \in I \mid \rhd_a \in I \mid \\ & \neg\varphi \mid \varphi \vee \varphi \mid \varphi\, U\, \varphi \mid X\varphi \quad (a \in \Sigma), \end{aligned}$$

The boolean operators, $X\varphi$, and $\varphi U \psi$, are interpreted as before in the untimed setting. The proposition a denotes that the event currently observed is a. Real-time constraints can be checked using $\lhd_a$ and $\rhd_a$. $\lhd_a \in I$ is the operator which measures the time elapsed since the last occurrence of a, and $\rhd_a \in I$ is the operator which predicts the next occurrence of a, both saying that this is within a timed interval I. For example, $G(\rhd_a \in [0, 5])$ requires an event a to occur again and again with a delay of at most 5 time units. Using these dedicated operators, we are now able to explicitly reason about real-time systems emitting real-time events to satisfy their respective deadlines.

In [2], we introduced the 3-valued variant of TLTL, which follows the same approach taken for LTL_3. Furthermore, we have also described how to generate for a given TLTL_3-formula a corresponding monitor function that reads events and outputs whether the events seen so far yield *true*, *false*, or just *inconclusive*.

4. Runtime diagnosis

In principle, diagnosis in our framework is based on the formal theory of *consistency-based diagnosis* introduced first by Reiter [24] and roughly at the same time, but independently under the name of *model-based diagnosis* by de Kleer and Williams [6].

4.1. Background: first-order diagnosis

From the diagnosis point of view, a system is a combination of a finite set of *components*, denoted by $COMP$. The components are considered as atomic entities, meaning that diagnosis will determine a subset of $COMP$ to be faulty, but—as expected—does not yield the actual "bug" within a component, e. g., division by zero or stack overflow. However, such a set of components can be of almost arbitrary granularity. Depending on the properties of the system to be diagnosed, $COMP$ may refer to, say, Java threads, user session objects within a web-server application, or even physical entities such as smart sensors, actuators, or entire nodes/CPUs of a computer network.

The overall system behaviour is then modelled in terms of the components' behaviours and their causality. In [24] and [6], first-order logic is used to describe the behaviour of a system. More specifically, first-order logic where the components in $COMP$ are used as (uninterpreted) constants is employed. Furthermore, a special predicate, AB, is used to denote that a component is abnormal; that is, presents a behaviour which is different to its specified or intended behaviour.

A *system* is then represented as a tuple $S = (SD, COMP)$, where $COMP$ is a finite set of *components* and SD constitutes a finite set of first-order sentences over the signature containing $COMP$, comprising the *system's description*.

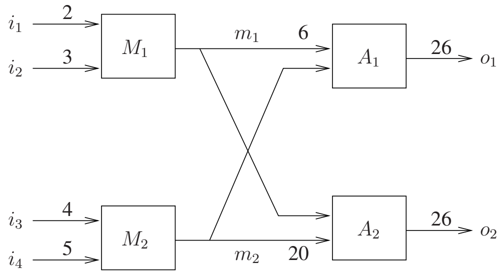

Figure 3. Application with four components.

Let's consider the application depicted in Fig. 3. We assume M_1, M_2 to be multiplicators, and A_1, and A_2 to be adders. The set of components is thus, $COMP = \{M_1, M_2, A_1, A_2\}$. For a multiplicator, the output is the product of its two inputs, *unless it does not work correctly*. We can model this fact by the formula

$$
\begin{aligned}
&mult(X) \wedge \neg AB(X) \Rightarrow \\
&\qquad (output_1(X) = input_1(X) \times input_2(X)).
\end{aligned}
$$

Thus, the crucial idea in the description above is to add the predicate $\neg AB(X)$, denoting that X is not *abnormal*, as a premise to the formula describing the correct functional behaviour. Thus, if $\neg AB(X)$ evaluates to true, i. e., component X works correctly, the output is, indeed, the product of the inputs. If X is abnormal, i. e., $\neg AB(X)$ is false, nothing has to hold for the conclusive part.

Overall, the system description SD then comprises the following list of first-order sentences:

$$
\begin{aligned}
&mult(X) \wedge \neg AB(X) \Rightarrow \\
&\quad (output_1(X) = input_1(X) \times input_2(X)), \\
&mult(M_1), \\
&output_1(M_1) = input_1(A_1), \\
&input_1(M_1) = i_1, \\
&input_2(M_1) = i_2, \ldots,
\end{aligned}
$$

Notice, the above list is not complete, in that not all components are described. At this point, it shall suffice to see how the system modelling is generally done, and how causality within the system is defined (i. e., in terms of input-output relations).

Formally, an *observation* corresponds to a mapping of in- and outputs to actual values, e. g., denoted as $OBS = \{i_1 \mapsto 2, i_2 \mapsto 3, \ldots\}$. Observations may be consistent with the system description, or not (in case of an occurred failure).

Given the tuple (SD, $COMP$, OBS), a *diagnosis* is then defined as a *minimal* set $\Delta \subseteq COMP$ such that

$$
\begin{aligned}
&SD \cup OBS \cup \\
&\{AB(c) \mid c \in \Delta\} \cup \{\neg AB(c) \mid c \in COMP \backslash \Delta\}
\end{aligned}
\tag{1}
$$

is consistent, i. e., satisfiable. In other words, the components of a diagnosis are set to be abnormal, which make the implications of the system description in which the components are involved hold trivially. Note that, in general, for a given system description and observation, several diagnoses also of different cardinality might exist.

It follows that the only interpretation for a diagnosis Δ with $\Delta = \emptyset$ is that the system is working as expected. Coming back to our example, it is self-evident that substituting in OBS, the output mapping $o_1 \mapsto v$ with a value $v \neq 26$, will lead to the conclusion $\Delta = \{A_1\}$, i. e., the following holds: $\neg AB(M_1)$, $\neg AB(M_2)$, $\neg AB(A_2)$, and $AB(A_1)$.

However, the approach outlined above has a serious limitation for automation. It is well-known that satisfiability of first-order logic is undecidable. Thus, there exists no automatic procedure for computing diagnoses for arbitrary system descriptions.

In the original theory of consistency-based diagnosis [24], the problem is addressed by using alternative characterisations of diagnoses in terms of *conflict sets* and employing (possibly non-terminating or interactive) first-order theorem provers for sub-goals.

Formally, a conflict set for $(SD, COMP, OBS)$ is a set $\{c_i, \ldots, c_j\} \subseteq COMP$ with $1 \leq i \leq j$ such that

$$SD \cup OBS \cup \{\neg AB(c_i), \ldots, \neg AB(c_j)\} \qquad (2)$$

is inconsistent, i. e., not satisfiable. Thus, the assumption that the components c_i of a conflict set work correctly does not explain the (partial) observations. In other words, a conflict set is a super set of those components assumed faulty, such that an abnormal system behaviour can be explained.

Surely, assuming all components to be faulty makes the previous formula unsatisfiable. Hence, $\{M_1, M_2, A_1, A_2\}$ (i. e., all components are faulty) would be a conflict set for our example, given $o_1 \mapsto 27$, and that m_1, m_2 are not observable. Further, a conflict set for $(SD, COMP, OBS)$ is called *minimal*, iff no proper subset of it is a conflict set for $(SD, COMP, OBS)$ at the same time. That is, $\{A_1\}$ is a minimal conflict set, but not the only possible conflict set.

Then diagnoses are obtained by first determining an initial conflict set using a first-order theorem prover, and then subsequently unfolding the (minimal) sets using the so called *hitting set algorithm*.

Theorem proving for first-order logic is either manual (i. e., interactive) or possibly non-terminating if automated. The hitting set problem, also known as the *transversal problem*, is one of the key problems in the combinatorics of finite sets and known to be NP-complete (cf. [9]). Hence, this complex, two-fold approach is hardly suitable to be performed at runtime, let alone for reactive or embedded systems.

4.2. Diagnosis as a SAT-problem

Fortunately, using the automatically generated monitors described in Sec. 3, it is possible to reduce the problem of diagnosis to a satisfiability problem of propositional logic, as described below. Recall that satisfiability of propositional logic is decidable and, more importantly, often solvable efficiently. Thus, we can develop efficient algorithms for solving the diagnosis problem.

Using monitors, we abstract from details of the system. A monitor's duty is to check whether a sequence of events satisfies a certain safety property (see Fig. 4). Then, for diagnosis, we no longer rely on the comparison of the actual values transmitted over some channels, but just on the information whether everything works according to the specified properties, in the following denoted by an *ok* predicate. Correspondingly, a system description is reduced to a set of formulas describing the correctness of input-output behaviour.

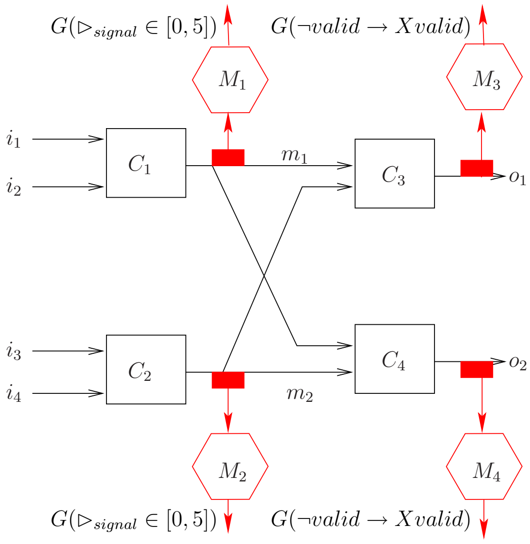

Figure 4. Application with four components and monitors.

Let's consider Fig. 4, depicting an abstract distributed system consisting of four components, $C_1, \ldots, C_4$, and monitors, $M_1, \ldots, M_4$, each of which observes a custom safety property. The system description SD can thus be reduced in terms of SD' as follows:

$$SD' = \left\{ \begin{array}{l} ok(i_1) \wedge ok(i_2) \wedge \neg AB(C_1) \Rightarrow ok(m_1), \\ ok(i_3) \wedge ok(i_4) \wedge \neg AB(C_2) \Rightarrow ok(m_2), \\ ok(m_1) \wedge ok(m_2) \wedge \neg AB(C_3) \Rightarrow ok(o_1), \\ ok(m_1) \wedge ok(m_2) \wedge \neg AB(C_4) \Rightarrow ok(o_2) \end{array} \right\},$$

where ok is the predicate denoting that a value or behaviour does not violate an expected behaviour, i. e., a corresponding safety property. Notice, for brevity, we have not modelled in this example any criteria for determining $ok(i_1), \ldots, ok(i_4)$.

As discussed in the section on runtime verification (Sec. 3), the monitors used for observing the details of the system, signal either true, false, or inconclusive. For diagnosis, we have to look for causes of violated properties. Thus, we can identify $true$ and ?. Therefore, in the following, assume a monitor to yield $true$ or $false$, where $true$ may also mean inconclusive.

The system description SD' in the form presented above can be converted into *conjunctive normal form*, denoted by $CNF(SD')$, in a straightforward manner using only polynomial time [16]. For example, we have for SD the following form:

$$CNF(SD') = \left\{ \begin{array}{l} \neg i1 \vee \neg i2 \vee AB(C1) \vee m1, \\ \neg i3 \vee \neg i4 \vee AB(C2) \vee m2, \\ \neg m1 \vee \neg m2 \vee AB(C3) \vee o1, \\ \neg m1 \vee \neg m2 \vee AB(C4) \vee o2 \end{array} \right\}.$$

When observing the system, we get for some input and some output values the information, whether the value is indeed ok or not.

Let us now assume that we have a monitor attached to all output channels of the application, except on m_1 and m_2 which remain unobservable (i. e., unknowns). Furthermore, assume we have the observations $OBS = \{i1, i2, i3, i4, \neg o1, o2\}$, meaning that the monitor observing o_1 has reported a failure.

In order to determine diagnoses explaining a monitor's result, we have to compute the (minimal) models for (1). In other words, the problem of determining diagnoses is now reducible to a propositional *satisfiability problem* (SAT), using $CNF(SD')$ rather than SD.

Although the *SAT-problem* is known to be NP-complete, there exist rather efficient algorithms which are able to determine the satisfiability of thousands of CNF-clauses and variables within seconds. Because of this, many other logic problems in computer science, such as model checking large state spaces, are often reduced to SAT-problems.

Using a SAT-solver we can now determine for the system and observations $S = (CNF(SD'), COMP, OBS)$ the sets of all possible sets, C_S, that explain $\neg o1$ by means of one, or many broken components:

$$C_S = \left\{ \begin{array}{l} \{C1, C2, C3, \neg C4\}, \\ \{C1, C2, \neg C3, \neg C4\}, \\ \{C1, \neg C2, C3, \neg C4\}, \\ \{C1, \neg C2, \neg C3, \neg C4\}, \\ \{\neg C1, C2, C3, \neg C4\}, \\ \{\neg C1, C2, \neg C3, \neg C4\}, \\ \{\neg C1, \neg C2, C3, \neg C4\} \end{array} \right\}.$$

Diagnoses, i.e. minimal sets showing satisfiability of (1), are the fourth and the last solution, meaning that either C_1 or C_3 is broken.

Diagnoses with minimal cardinality. Actually, we have to find minimal satisfying solutions of (1). While the SAT-problem is NP-complete, the so-called #SAT-problem is known to be in the much bigger class #P [19]. Therefore, we still face a complexity problem. We solved it by building our own SAT-solver, which is described in greater detail in [3]. In a nutshell, it works as follows. Assuming that components fail independently, it is very unlikely that those diagnoses are relevant diagnoses in which, e. g., all the components are marked faulty. Of course, this may happen, but (say) from experience or service reports of a certain system, we may assume that the most likely diagnoses are those where merely one or two components are marked faulty. Our custom solver component, named LSAT, reflects this knowledge in its main data structures and solving algorithm, in that it prunes the search space based on the *cardinality* of the AB-predicates. In other words, given a two-fault assumption, for example, LSAT would merely return solutions containing at most two faulty components. Other solutions are pruned from the search-space.

Using the monitors in combination with diagnosis, we have at hand a propositional, hence, very efficient mechanism for differentiating symptoms for a failure, i. e., $\neg ok(o_1)$, from actual causes, e. g., $AB(C_1)$, which is based upon the cardinality of the AB-predicates, rather than solutions obtained by using a theorem prover or the HS-algorithm, for instance.

5. Implementation and results

Our runtime reflection framework currently consists of the core components for performing runtime verification and diagnosis, i. e., we have implemented and provide freely the logging layer, monitoring layer, as well as the diagnosis layer which hints to faulty system components in the case of an occurred error. In the following, we therefore give a brief overview on the respective technicalities regarding their implementation.

5.1. Logging and monitoring

Currently, we provide an extensive and versatile logging layer for distributed and multi-threaded C++-applications. The logging layer offers two separate interfaces: first, a logging interface which is used by the observed application to generate outside-visible system events, and second, a configuration interface which allows to customise the logging and monitoring facilities in an arbitrary manner. To integrate a custom application written in C++ with the logging layer, it is necessary to annotate the application's code. Our

Table 1. Modified ISCAS'89 benchmarks under the n-fault assumption.

				∞-fault		5-fault	
Name:	$\#COMP$:	#Var.:	#Cl.:	#Steps:	CPU:	#Steps:	CPU:
s208.1	66	122	389	84	0.17 sec	60	0.25 sec
s298	75	136	482	27	0.11 sec	58	0.32 sec
s444	119	205	714	20	0.18 sec	105	0.91 sec
s526n	140	218	833	–	timeout	295	0.23 sec
s820	256	312	1,335	–	timeout	562	0.59 sec
s1238	428	540	2,057	38	0.97	262	0.21 sec
s13207	2,573	8,651	27,067	–	timeout	17	0.57 sec
s15850	3,448	10,383	33,189	–	timeout	41	0.17 sec
s35932	12,204	17,828	60,399	2,339	11.16 sec	29	0.21 sec

logging layer provides a large number of annotations for this purpose, for example, to log certain method entries and exits or unexpected exceptions.

Based on this logging layer, we also provide with our framework a dedicated generator-tool to automatically create a monitor based on a specification written in LTL. The generated monitor is then provided in terms of a C++-class, which implements the main communication interface employed in the logger layer.

5.2. Diagnosis

Diagnosis in the runtime reflection framework is performed by employing a custom SAT-solver, optimised for consistency-based diagnosis as outlined in Sec. 4. Instead of determining the minimal hitting sets of all possible conflicts, we employ a data structure that provides diagnoses based on the minimal cardinality of abnormal components (cf. [3]). In other words, only those diagnoses are computed, which contain at most n faulty components, where n is a variable that can be chosen by the user, e. g., based on known probabilities of failure, or failure rates. We referred to this earlier as the n-fault assumption, which constitutes the pruning criterion of the data structure representing all the possible supersets of diagnoses. For example, a 2-fault assumption indicates that all possible diagnoses are omitted, in which more than two components would be declared faulty.

Technically, the diagnostic engine obtains from the monitors information on the status of the components determined via safety properties. In terms of the overall framework, this allows for an efficient analysis, in that we trigger diagnosis only if at least one monitor has detected an abnormal behaviour in some component of the system under scrutiny. Alternatively, the diagnostic engine can be used stand-alone, e. g., for off-line analysis of arbitrary systems.

We have validated this approach experimentally by inducing random faults in large micro-chip designs with tens of thousands of clauses and variables, and have restricted ourselves to a five-fault assumption. Notice, from the diagnostic point of view alone, it is irrelevant as to whether the system to be diagnosed is a micro-chip, or a large distributed system, as long as an adequate system model for diagnosis is available.

The results of the computations were almost instantaneous, i. e., the search never occupied more than a second on a standard PC (i686, ca. 2 GHz, standard Linux kernel). Without the optimisation, several seconds were occupied and occasionally no solution found at all (see Table 1).

6. Conclusions and future work

Our framework for runtime analysis as we have presented it in this paper provides tools and methods that enable distributed reactive systems to reflect upon their system status at runtime. Due to the layered architecture and the efficient combination and realisation of different techniques for reasoning about such systems, i. e., runtime verification and subsequent model-based diagnosis, we avoid some typical pitfalls that exist in analysing distributed systems at runtime when using more "monolithic" methods as described, e. g., in Sec. 1. Foremost, our component-oriented approach triggers diagnosis specifically at the occurrence of a fault, which avoids a continuous computational effort on the diagnoser's side. Additionally, the use of independent and local monitors in order to observe specific components, avoids an expensive communication penalty in that no extra diagnostic messages need to be exchanged between the respective monitors in order to come to a verdict regarding a system's overall status.

We have successfully implemented the ideas presented in this paper and are currently in the process of streamlining the entire architecture for ease of integration and further extensibility towards recovery measures, for instance. The latter were, on purpose, not intensively dealt with in this paper, since they constitute highly domain-specific knowledge and methods, which are not necessarily applicable to all real-time or reactive systems alike. Consider, for instance, the differences between distributed control systems and business information systems.

The results we presented for the diagnosis component, however, hint to the scalability of our approach and show the potential for deployment even in resource-bounded environments such as embedded systems, where it is often even more difficult to differentiate between symptoms of a failure and its cause, since access to the system's internals is often limited. Moreover, the ability to reason about gray-box systems (i. e., reasoning in the presence of unknowns), as we have discussed earlier, is additionally interesting for such settings.

Finally, we are developing and provide the runtime reflection framework free and under an open-source license, namely the GNU General Public License (see `http://runtime.in.tum.de/`), thus enabling wide-spread deployment in various settings, and to provide a platform for future add-ons and developments possibly even by a third party.

References

[1] R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2):253–273, 1999.

[2] O. Arafat, A. Bauer, M. Leucker, and C. Schallhart. Runtime verification revisited. Technical Report TUM-I0518, Technische Universität München, 2005.

[3] A. Bauer. Simplifying diagnosis using LSAT: a propositional approach to reasoning from first principles. In *Proc. CP-AI-OR*, volume 3524 of *LNCS*, Prague, Czech Republic, June 2005. Springer-Verlag.

[4] M. Broy. Mathematical system models as a basis of software engineering. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 1995.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[6] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *AI*, 32(1):97–130, 1987.

[7] D. Drusinsky. The temporal rover and the ATG rover. In *SPIN*, pages 323–330, 2000.

[8] D. D'Souza. A logical characterisation of event clock automata. *Int. Journ. Found. Comp. Sci.*, 14(4):625–639, Aug. 2003.

[9] T. Eiter and G. Gottlob. Hypergraph transversal computation and related problems in logic and AI. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *JELIA*, volume 2424 of *LNCS*, pages 549–564. Springer, 2002.

[10] C. A. Ericson, II. *Hazard Analysis Techniques for System Safety*. John Wiley and Sons Inc., Aug. 2005.

[11] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, New York, NY, USA, 1985.

[12] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. *Electr. Notes Theor. Comp. Sci.*, 55(2), 2001.

[13] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Journ. Softw. Tools for Tech. Transf.*, 2004.

[14] R. Isermann. Model-based fault detection and diagnosis: status and applications. In *Proceedings of the 16th IFAC Symposium on Automatic Control in Aerospace*, St. Petersburg, Russia, June 2004.

[15] H. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[16] A. Nonnengart and C. Weidenbach. Computing small clause normal forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science B.V., 2001.

[17] M. Nyberg. *Model Based Fault Diagnosis: Methods, Theory, and Automotive Engine Applications*. PhD thesis, Linköpings Universitet, June 1999.

[18] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New York, Jan. 1985. ACM.

[19] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.

[20] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, Oct. 31–Nov. 2 1977. IEEE Computer Society Press.

[21] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current trends in concurrency. Overviews and tutorials*, pages 510–584, New York, NY, USA, 1986. Springer-Verlag.

[22] J.-F. Raskin and P.-Y. Schobbens. State clock logic: A decidable real-time logic. In O. Maler, editor, *HART*, volume 1201 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 1997.

[23] J.-F. Raskin and P.-Y. Schobbens. The logic of event clocks - decidability, complexity and expressiveness. *Journ. of Autom. Lang. and Comb.*, 4(3):247–286, 1999.

[24] R. Reiter. A theory of diagnosis from first principles. *AI*, 32(1):57–95, 1987.

[25] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 418–427. IEEE Computer Society, 2004.

[26] D. H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. ASQ Quality Press, second edition, Apr. 2003.

[27] W. E. Vesely et al. Fault tree handbook. Technical Report NUREG-0492, Systems and Reliability Research, Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Commission, Washington, DC, 1981.

[28] P. Wolper. The meaning of "formal": From weak to strong formal methods. *STTT*, 1(1–2):6–8, 1997.