Using LNT Formal Descriptions for Model-Based Diagnosis

Birgit Hofer[1] and **Radu Mateescu**[2] and **Wendelin Serwe**[2] and **Franz Wotawa**[1]
[1]Institute for Software Technology, TU Graz, Inffeldgasse 16b/II, 8010 Graz, Austria
e-mail: {bhofer,wotawa}@ist.tugraz.at
[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, 38000 Grenoble, France
e-mail: {Radu.Mateescu,Wendelin.Serwe}@inria.fr

Abstract

Providing models for model-based diagnosis has always been a challenging task. There has never been an agreement on an underlying modeling language, making it almost impossible to share models within our community. In addition, there are other domains like formal methods or model-based testing relying on system models for formal verification and automated test case generation. Although, there we face the situation of different modeling languages as well, the question remains whether it is possible to re-use these models in the context of model-based diagnosis. In this paper, we elaborate on this question and show how models written in LNT can be used for fault localization only requiring simple modification. This allows re-using formal method's models for diagnosis directly. Besides discussing the underlying principles, we also present a use case showing the applicability of the methods.

1 Introduction

Despite the fact that model-based diagnosis offers a lot of advantages compared to other approaches of diagnosis, its use in practice, despite running as part of prototype implementations and case studies, is somehow limited. One reason behind is that modeling in general is a non trivial task and that there is a lack on commonly agreed modeling languages that are capable of providing the right means for modeling for diagnosis. This includes capturing temporal behavior and also dealing with the right level of abstraction.

In many industrial applications being able to handle time appropriately as well as models that closely capture physical properties is essential and for some of those issues a specialized diagnosis procedure has already been presented, including diagnosis for communication systems (e.g. [1] or [2]) or the use of modeling languages like Modelica for extracting models to be used for model-based diagnosis (see [3]).

In this paper, we tackle the challenge of providing models for model-based diagnosis. But instead of relying on models used for simulation, we elaborate on the use of specification languages for diagnosis. Formal specification languages have been developed for modeling systems with the primary purpose of validation, which can be done using testing or formal verification. In case of a system model, either test cases can be automatically extracted from the model or additional properties can be checked using the system model. Testing as a task can be seen as a falsification step for systems whereas formal verification as a proof whether certain properties are holding generally.

The objective behind this paper is to show on the example of LNT [4], that these formal specification languages can not only be used for verification, but also for diagnosis of systems. If formal models in LNT can be effectively used for diagnosis, first we would be able to re-use these LNT models in a diagnosis setting, and second we could use LNT as a general modeling language that can be used for simulation, verification, and also diagnosis of the same system. For this purpose, we introduce a general wrapper component that adds diagnosis capabilities to any LNT model of the same component. Hence, we do not use the original LNT model for diagnosis but an extended model where each component that should be considered for diagnosis has to be replaced with its corresponding wrapper component. Such a wrapper component cannot only capture the unknown faulty behavior, which is used in consistency-based diagnosis [5] but also fault models used in abductive diagnosis [6].

Besides discussing the methodologies behind the application of LNT to diagnostic reasoning, we also report on applying diagnosis to a model of the data encryption standard where we were able to localize faults using the basic concept of wrapper components for diagnosis.

This paper is organized as follows: To be self contained, we first discuss preliminaries including model-based diagnosis and the basic concepts behind LNT. Afterwards, we introduce the methodology behind using LNT models for diagnosis including the concept of wrapper components and obtaining diagnosis candidates from them. Furthermore, we report on the results obtained from a case study based on the data encryption standard where we manually introduced faults. Finally, we discuss related research and conclude the paper.

2 Preliminaries

In this section, we discuss the basic definitions of model-based diagnosis and formal methods in order to be self-contained. We mainly focus on formal methods and introduce the modeling language LNT (formerly called *LOTOS New Technology*) and CADP (*Construction and Analysis of*

*Institute of Engineering Univ. Grenoble Alpes

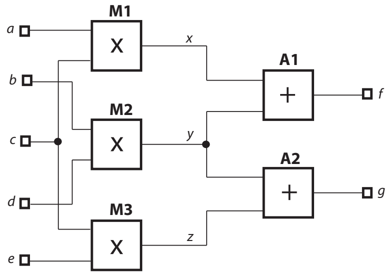

Figure 1: The classical d74 circuit.

Distributed Processes) tools available[1]. For the sake of simplicity, we make use of the classical d74 circuit from Figure 1 often used in diagnosis literature, e.g., see [7], to explain both model-based diagnosis and formal methods based on LNT.

2.1 Consistency-based diagnosis

We briefly discuss the basic definitions behind model-based diagnosis, and there consistency-based diagnosis in particular, which are based on Reiter's seminal work [5]. We start defining a diagnosis system comprising a model SD and the set of components $COMP$. The idea here is to allow only elements of $COMP$ to be working as expected or faulty. In order to make the information regarding a component's health state explicit, we introduce a predicate $Ab(C)$ for each component C from $COMP$ stating that C behaves abnormally. The model SD itself covers the structure of a system and the behavior of each component. In case of consistency-based diagnosis, only the correct behavior of components is covered. This can be formalized using rules of the form $\neg Ab(C) \rightarrow$ *behavioral description*. When modeling using implications, we do not restrict the faulty behavior of components. In particular, a component may behave correctly even in case its health state indicates that it is faulty. For a detailed discussion on fault models we refer to [8].

Definition 2.1 (Diagnosis system)**.** *A diagnosis system is a tuple $(SD, COMP)$ where SD is a set of logical sentences describing the structure and behavior of the system, and $COMP$ the set of system components.*

Example 1. *For the d74 circuit, we are able to specify the behavior of the adder and multiplier components as follows:*

$$ADD(C) \rightarrow \big(\neg Ab(C) \rightarrow out(C) = in_1(C) + in_2(C)\big)$$
$$MULT(C) \rightarrow \big(\neg Ab(C) \rightarrow out(C) = in_1(C) * in_2(C)\big)$$

The structure of the circuit comprises a definition of the components and their connections. In this particular case $COMP = \{M1, M2, M3, A1, A2\}$.

$$\begin{gathered} MULT(M1) \wedge MULT(M2) \wedge MULT(M3) \wedge \\ ADD(A1) \wedge ADD(A2) \wedge \\ out(M1) = in_1(A1) \wedge out(M2) = in_2(A1) \wedge \\ out(M2) = in_1(A2) \wedge out(M3) = in_2(A2) \end{gathered}$$

The model of the d74 circuit comprises all the described logical rules. □

[1]`http://cadp.inria.fr`

From a diagnosis system we immediately come to a diagnosis problem, when adding observations.

Definition 2.2 (Diagnosis problem)**.** *Given a diagnosis system $(SD, COMP)$ and a logical sentence OBS describing the given observations. The tuple $(SD, COMP, OBS)$ is a diagnosis problem.*

Example 2. *(cont. Ex. 1) For the d74 circuit we may use the following observation set OBS:*

$$\begin{gathered} in_1(M1) = 2 \wedge in_2(M1) = 3 \wedge \\ in_1(M2) = 2 \wedge in_2(M2) = 3 \wedge \\ in_1(M3) = 3 \wedge in_2(M3) = 2 \wedge \\ out(A1) = 10 \wedge out(A2) = 12 \end{gathered}$$

Obviously, there is a fault in the circuit, because the output of adder $A1$ needs to be 12 and not 10 as observed when assuming all components to work as expected. □

Given a diagnosis problem, a diagnosis should explain deviations between expected values at any input and output as well as intermediate connection between components, and given observations. In consistency-based diagnosis a diagnosis is defined as a set of components that when assumed to behave abnormally and all other components are expected to work correctly, will not contradict any observation when using the underlying system model. Formally, diagnoses are defined as follows:

Definition 2.3 (Diagnosis)**.** *Given a diagnosis problem $(SD, COMP, OBS)$. A set of components $\Delta \subseteq COMP$ is a diagnosis if and only if $SD \cup OBS \cup \big\{Ab(C) \mid C \in \Delta\big\} \cup \big\{\neg Ab(C) \mid C \in COMP \setminus \Delta\big\}$ is satisfiable.*

In this definition a diagnosis needs not to be minimal. We define a minimal diagnosis as a diagnosis where none of its subsets is itself a diagnosis accordingly to Definition 2.3.

Example 3. *(cont. Ex. 2) For the given observations and the d74 model, the four sets $\{M1\}, \{A1\}, \{M2, M3\}$, and $\{M2, A2\}$ are all minimal diagnoses and there are no other minimal diagnoses.* □

The dual concept of diagnoses are conflicts, i.e., a set of components that, when assumed to behave correctly, together with the model SD contradict the given observations OBS.

Definition 2.4 (Conflict)**.** *Given a diagnosis problem $(SD, COMP, OBS)$. A set of components $CO \subseteq COMP$ is a conflict, if and only if $SD \cup OBS \cup \big\{\neg Ab(C) \mid C \in CO\big\}$ is a contradiction.*

Example 4. *(cont. Ex. 3) For the d74 circuit we obtain 2 minimal conflicts, i.e.: $\{M1, M2, A1\}$, and $\{M1, M3, A1, A2\}$.* □

Reiter [5] showed that there is a close relationship between diagnoses and conflicts. In particular, every minimal diagnosis is a minimal hitting set of all conflicts. Reiter also introduced an algorithm computing such hitting sets where conflicts are computed during computation. Greiner *et al.* [9] provided a corrected version of Reiter's diagnosis algorithm. However, there are many other algorithms available for computing diagnoses. Some are based on conflicts whereas the others compute diagnoses directly from the model and the given observations. Nica *et al.* [10] introduced an empirical evaluation of the runtime of different diagnosis algorithms. Note that in this paper, we make use of algorithms for computing diagnoses directly from models.

2.2 Formal methods

LNT [11] culminates a 30-year effort [4] aimed at supplementing the international standard LOTOS [12] with language features borrowed from classical programming languages in order to enhance its user-friendliness and allow for a wider industrial dissemination. LNT is firmly rooted in concurrency theory: its operational semantics is defined as an LTS (*Labeled Transition System*) and its composition operators are compatible with behavioral equivalences (bisimulations).

In general, the behavior of an LNT model is defined as the parallel composition of processes communicating and synchronizing only by multiway rendezvous [13, 14]. Each of these processes is described with usual programming constructs (assignments, if-then-else, loops, etc.) and can manipulate data values and complex data structures (such as lists and trees).

LNT is the principal modeling language supported by the CADP (*Construction and Analysis of Distributed Processes*) toolbox [15], which provides an extensive set of languages and tools assisting the whole design process: compilation and rapid prototyping, interactive and guided simulation, LTS generation, equivalence and model checking, test case generation, and performance evaluation. Among these tools, the most useful for diagnosis are the LNT compilers, the equivalence checker BISIMULATOR, the model checker EVALUATOR [16], and the SVL language [17] for describing verification scenarios. There also exists tools for (distributed) code generation and test case extraction. A noteworthy feature of BISIMULATOR and EVALUATOR is that these tools operate on the fly, i.e., they only explore the part of the model required to obtain a result.

LNT and CADP have been used for many case studies in various domains[2]: avionics, cloud computing, distributed algorithms, hardware design, human-computer interaction, industrial systems, etc.

3 Using LNT for diagnosis

To use an LNT model for diagnosis, it must be parameterized to enable the selection of the set of components that should behave according to the considered fault model. Concretely, this implies

1. to wrap all individual components inside wrapper processes with a Boolean parameter to select between normal and faulty behavior and
2. to add these parameters to the overall system.

3.1 Wrapping individual components

To illustrate the wrapping of a component, consider the following LNT model of an adder, such as $A1, A2$ in the d74 circuit.

```
process ADDER [IN1, IN2, SUM: NAT_C] is
   var in1, in2, result: Nat in
      loop
         par
            IN1 (?in1)
         || IN2 (?in2)
         end par;
         result := in1 + in2;
         SUM (result)
      end loop
   end var
end process
```

[2] http://cadp.inria.fr/case-studies

It repeatedly (instruction `loop ... end loop`) waits for two natural numbers (`in1` and `in2`) on its gates `IN1` and `IN2`, computes the result as soon as both inputs are available, and then outputs the result on its gate `SUM` (in a rendezvous, inputs are described by `?`x, where x is a variable that will be assigned by the rendezvous). The inputs are read in parallel (instruction `par ... end par`), i.e., without any constraint on the order. All gates are of channel type `NAT_C`, specifying that a natural number is communicated during a rendezvous.

The process `ADDER` can be wrapped inside a process `ADDER_WRAP` with a Boolean parameter `faulty` (corresponding to the predicate Ab), which, using an **if-then-else**, chooses between a call to the original process `ADDER` and a faulty version. The faulty version is the same as the body of the process `ADDER` excepting for the computation of the result, in which the output of a concrete value is replaced by a nondeterministic assignment (instruction "`:= any Nat`"), constrained by a predicate "`P (in1, in2, result)`" returning **true** if and only if the wrapped process should allow the output `result` for the inputs `in1` and `in2`. For instance, `P` could be used to specify that certain bits of the result are forced to a constant.

```
process ADDER_WRAP [IN1, IN2, SUM: NAT_C]
                   (faulty: Bool) is
   if faulty then
      loop
         var in1, in2, result: Nat in
            par
               IN1 (?in1)
            || IN2 (?in2)
            end par;
            result := any Nat
               where P (in1, in2, result);
            SUM (result)
         end var
      end loop
   else
      ADDER [IN1, IN2, SUM]
   end if
end process
```

To represent the most generic failure model, where predicate `P` always returns **true**, i.e., where any output can be nondeterministically chosen, the wrapper process can be simplified by removing the local variables, leaving all rendezvous unconstrained ("`?any Nat`"):

```
process ADDER_WRAP_ND [IN1, IN2, SUM: NAT_C]
                      (faulty: Bool) is
   if faulty then
      loop
         par
            IN1 (?any Nat)
         || IN2 (?any Nat)
         end par;
         SUM (?any Nat)
      end loop
   else
      ADDER [IN1, IN2, SUM]
   end if
end process
```

This approach can be generalized to arbitrary processes. Indeed, it is sufficient to copy the original code and modify any constraints on the rendezvous according to the chosen fault model — in the extreme case removing the constraints completely.

3.2 Analyzing faulty configurations

The behavior of the whole system is obtained by composing all wrapper processes in parallel and synchronizing them according to the system architecture. The LNT `MAIN` process describing the d74 circuit from Figure 1 is shown below. Each of the five components has a synchronization interface consisting of its input and output gates. The gates corresponding to the interaction of the system with its environment (e.g., the entries `IN1`, `IN2` of multiplier `M1` or the output `OUT2` of adder `A2`) are kept visible, whereas the gates that connect and synchronize components (e.g., the output `C1` of `M1` connected to the first input of `A1`), are abstracted away (i.e., hidden by the **hide** operator) in the final system: they are not in the list of gate parameters of `MAIN`.

```
process MAIN [IN1, IN2, IN3, IN4, IN5,
              OUT1, OUT2: NAT_C]
             (f1, f2, f3, f4, f5: Bool,
              i1, i2, i3, i4, i5: Nat) is
   hide C1, C2, C3: NAT_C in
      par
         IN1, IN2, IN3, IN4, IN5 ->
            IN1 (i1); IN2 (i2); IN3 (i3);
            IN4 (i4); IN5 (i5); stop
      || IN1, IN3, C1 -> (* M1 *)
            MULTI_WRAP [IN1, IN3, C1] (f1)
      || IN2, IN4, C2 -> (* M2 *)
            MULTI_WRAP [IN2, IN4, C2] (f2)
      || IN3, IN5, C3 -> (* M3 *)
            MULTI_WRAP [IN3, IN5, C3] (f3)
      || C1, C2 ->       (* A1 *)
            ADDER_WRAP [C1, C2, OUT1] (f4)
      || C2, C3 ->       (* A2 *)
            ADDER_WRAP [C2, C3, OUT2] (f5)
      end par
   end hide
end process
```

The health states of individual components (i.e., the values of the predicate Ab) are given by the Boolean parameters `f1`, ..., `f5` of the `MAIN` process, which are used to instantiate the `faulty` arguments of the wrapper processes. The values of inputs, injected into the system by the first behavior of the **par** operator, are given by the `i1`, ..., `i5` parameters of `MAIN`. By varying these parameters of the model, various faulty configurations of the system can be explored.

In the CADP setting, the consistency-based diagnosis approach, i.e., checking whether a set of components $\Delta \subseteq COMP$ is a diagnosis for a diagnosis problem $(SD, COMP, OBS)$ can be carried out as follows:

1. model the system structure SD and the behavior of individual components $COMP$ in LNT,
2. instantiate the system, specifying a component C as faulty (via the corresponding parameter) if and only if C belongs to Δ,
3. represent the observations OBS as temporal formulas (in MCL [16]) or sequences of events (i.e., a particular kind of LTS), and
4. determine the presence of observations in the considered system configuration using on-the-fly verification techniques, e.g., model checking (with EVALUATOR) or checking inclusion modulo equivalence relations (with BISIMULATOR).

Note that the instantiation (step 2) and the use of on-the-fly verification techniques (step 4) help in handling models with a large state space only a small fragment of which is reachable and necessary to inspect.

Once the LNT model of the system and components is available, the analysis of various faulty configurations of the system can be readily performed using SVL [17] scripts invoking the appropriate CADP tools. For the d74 circuit, we can represent the observation set given in Example 2 by the following event sequence in the SEQ format of CADP (where each line corresponds to the label of transition):

```
"IN1 !2"
"IN2 !3"
"IN3 !3"
"IN4 !2"
"IN5 !2"
"OUT1 !10"
"OUT2 !12"
```

Assuming the observation sequence is stored in a file `"obs.seq"`, the following SVL statements verify the inclusion of the sequence (modulo the preorder of branching bisimulation) in the models of the healthy system and of the faulty system with diagnosis $\{M2, M3\}$ from Example 3:

```
% I1=2; I2=3; I3=3; I4=2; I5=2
branching comparison
   "obs.seq" <= "MAIN(false,false,false,
false,false,$I1,$I2,$I3,$I4,$I5)" ;
branching comparison
   "obs.seq" <= "MAIN(false,true,true,
false,false,$I1,$I2,$I3,$I4,$I5)" ;
```

Note the usage of shell-script instructions (lines starting with a `%`) to initialize the shell-script variables `I1`, ..., `I5`, which are subsequently used to feed the input of the MAIN processes representing the two system configurations. The health states of the components are set by giving appropriate values to the Boolean parameters `f1`, ..., `f5` of the MAIN processes. The results of the two verifications above show that the observation sequence is absent in the healthy model and present in the faulty one.

The same verification can be carried out using on-the-fly model checking, by encoding the existence of the observation sequence as a weak possibility modality "**<<** ... **>> true**" in MCL [16] and then evaluating it on a given system configuration. The SVL statement (note the inlined MCL formula) below performs this check (which yields a positive verdict, as expected) on the faulty configuration $\{M1\}$ from Example 3.

```
property FAULTY_M1_OBS is
   "MAIN(true,false,false,false,false,
$I1,$I2,$I3,$I4,$I5)" |= with evaluator4
      <<
         "IN1 !2" .
         "IN2 !3" .
         "IN3 !3" .
         "IN4 !2" .
         "IN5 !2" .
         "OUT1 !10" .
         "OUT2 !12"
      >> true ;
   expected TRUE
end property ;
```

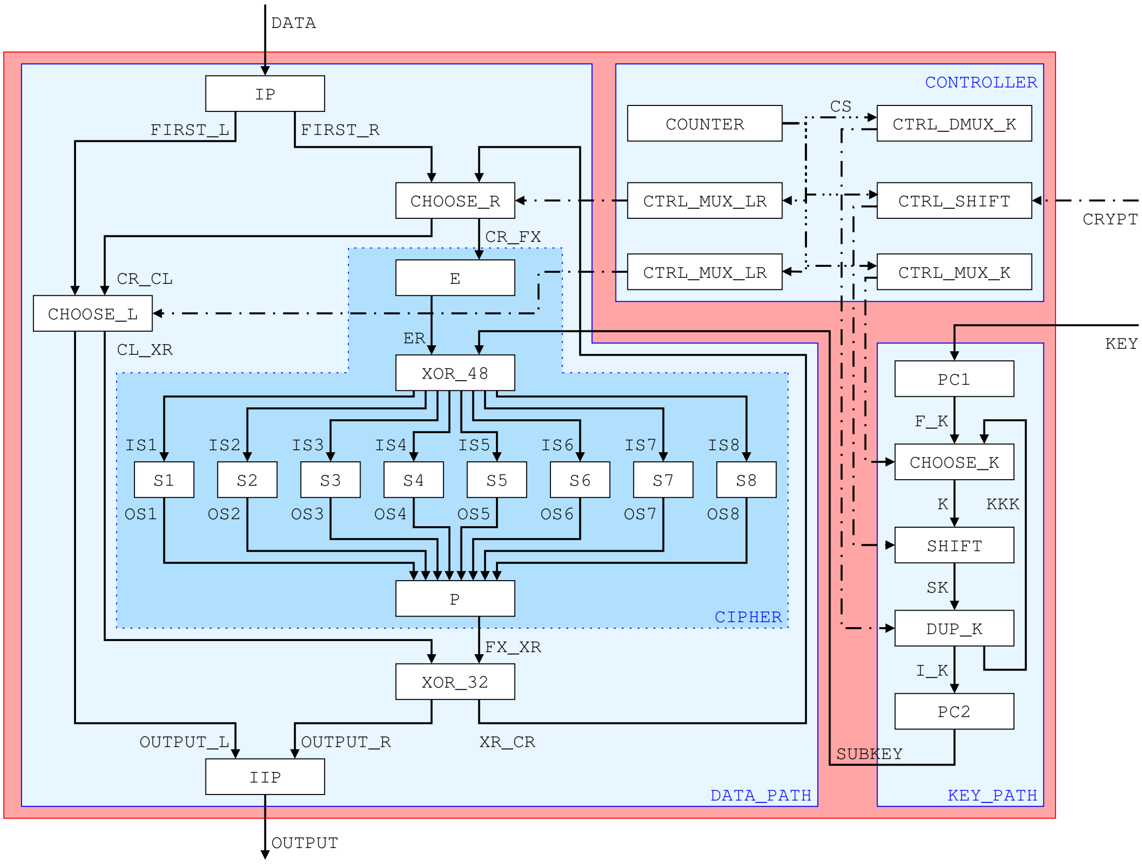

Figure 2: Architecture of the DES asynchronous circuit

This consistency checking approach can be easily integrated into classical diagnosis algorithms, such as HSDAG (*Hitting Set Directed Acyclic Graph*) [5], either by encoding the diagnosis algorithm as an SVL script, or by connecting an existing implementation of it with the CADP toolbox, by implementing consistency checks by system calls to CADP's equivalence and model checkers operating on the LNT model of the system under diagnosis.

4 Case study: asynchronous DES circuit

To study the feasibility and scalability of the approach, we experimented with the LNT model of an asynchronous implementation of the DES (Data Encryption Standard) [18]. This model is interesting, because it is publicly available as a demo example of the CADP toolbox[3], because it is complex (more than twenty processes and a corresponding LTS with several million states and transitions), and because cryptographic algorithms should challenge fault localization, as they aim to hide internal computations.

In a nutshell, the DES is a block-cipher taking three inputs: a Boolean indicating whether encryption or decryption is requested, a 64-bit key, and a 64-bit block of data. For each triple of inputs, the DES computes the 64-bit (de-)crypted data, performing sixteen iterations of the same cipher function, each iteration with a different 48-bit sub-key extracted from the 64-bit key.

[3] `ftp://ftp.inrialpes.fr/pub/vasy/demos/demo_38`

S_1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Figure 3: Table specification of the S-box S_1

The DES is specified as a data-flow diagram [19], which translates smoothly to the architecture shown on Figure 2. Roughly, a `CONTROLLER` schedules the flow of the key (respectively, the data) through the `KEY_PATH` (respectively, `DATA_PATH`). The main computation is performed by the cipher function `CIPHER` in the `DATA_PATH`.

The principal elements of `CIPHER` are so-called S-boxes (noted `S1`, ..., `S8` on Figure 2), which compute for a 6-bit input vector a 4-bit output vector. Given that each S-box is specified by a table with four rows and 16 columns (see Figure 3, taken from [19, Appendix 1], for S_1), human errors in implementing these tables are highly probable. In LNT, these tables are encoded as two-dimensional arrays (in LNT, constants are represented by functions without arguments)

```
function S1 : S_BOX_ARRAY is
return
   S_BOX_ARRAY
     (ROW (14,  4, 13,  1,  2, 15, 11,  8,
            3, 10,  6, 12,  5,  9,  0,  7),
      ROW ( 0, 15,  7,  4, 14,  2, 13,  1,
           10,  6, 12, 11,  9,  5,  3,  8),
      ROW ( 4,  1, 14,  8, 13,  6,  2, 11,
```

```
            15, 12,  9,  7,  3, 10,  5,  0),
       ROW (15, 12,  8,  2,  4,  9,  1,  7,
             5, 11,  3, 14, 10,  0,  6, 13))
end function
```

The LNT model of the DES has been validated in several ways (see the SVL script of the CADP demo for details). In particular, several properties expressing the correct ordering of the sixteen iterations have been expressed as MCL formulae and checked with EVALUATOR. Also, a prototype implementation was derived from the LNT model and used to check the correctness by comparing the result to known results of several reference implementations. These verification steps are described in [18] and can be replayed by executing the SVL script included in the CADP demo.

To obtain an incorrect output, we falsified the model of the DES by modifying one entry in one of the S-boxes such that a single bit of output was flipped. Then we studied whether using wrapper processes (on the original, correct model of the DES) could identify the S-box responsible for the incorrect output. For the wrappers, we considered the most generic fault model: hence, a faulty S-box may return any 4-bit vector. The wrapper process for the S-Box S_1 is:

```
process S1_WRAPPER [INPUT: C6, OUTPUT: C4]
                   (faulty: Bool) is
   if faulty then
      loop
         INPUT (?any BIT6);
         OUTPUT (?any BIT4)
      end loop
   else
      S1 [INPUT, OUTPUT]
   end if
end process
```

A faulty S-box may produce 16 possible outputs (rather than a single one), so that, due to the sixteen iterations of the DES, the complete model would have $16^{16} = 2^{64}$ possible outputs. Because this is clearly too large, we simplified the model of the DES to perform only a single iteration, so that the state space becomes manageable. For both, the correct and incorrect model, the corresponding LTS has 79,416 states[4] and 513,940 transitions (64 states and 82 transitions after reduction with strong bisimulation), and for a model with a single activated wrapper, the corresponding LTS has 473,316 states and 3,204,445 transitions (1000 states and 2551 transitions after reduction with strong bisimulation).

Checking the inclusion of the incorrect model in each of the models with a single faulty S-box identifies the S-box responsible for the incorrect output, because the inclusion holds only for the model where the (incorrect) S-box is faulty. Thus, there is no need to consider instances with multiple faulty S-boxes.

We used an SVL script to

1. generate and minimize the correct model, the incorrect model, and the eight models with one faulty S-box,
2. check that the correct and incorrect model are not branching bisimilar,
3. check that the correct model is included (modulo the preorder of branching bisimulation) in each of the models with a faulty S-box, and
4. check whether the incorrect model is included (modulo the preorder of branching bisimulation) in one of the models with a faulty S-box.

On a laptop with a Intel Core i5 M560 CPU at 2.67 Ghz and 8 MB of RAM, executing this SVL script takes about eleven minutes, the bunch of the time being spent in the generation and minimization step (i.e., step 1); the comparisons with BISIMULATOR (i.e., steps 2 to 4) only take seconds.

Experiments with other errors in the S-boxes, such as a copy-paste error (replace the definition of an S-box by the definition of another one), led to similar results, because in a single iteration each S-box is called only once so that only one error is visible.

5 Related Work

Shapiro was one of the first who introduced an automated software debugging approach in the 80's. Davis [20] and Reiter [5] proposed model-based diagnosis approaches to locate faults in hardware. In the 90's, Console *et al.* [21] applied model-based diagnosis to software, in particular logic programs. Bond [22] improved the work of Console *et al.*. In the late 90's, several researchers used the principles of model-based diagnosis to locate faults in programs written using sequential, concurrent, and functional programming languages [23–25]. We refer the interested reader to Wong *et al.*'s overview paper on software fault localization [26].

Pill and Quaritsch [27] proposed a scenario-based approach for diagnosing faults in formal LTL specifications. In contrast to our work, they support weak and strong fault models. Peischl *et al.* [28] proposed to use Modelica models to describe cyber-physical systems and to derive fault models from these models.

Another way of diagnosing faults in component-based systems is *blaming*, introduced by Goessler and Astefanoaiei [29] to determine the components responsible for errors in safety-critical, real-time systems. Starting from an execution trace violating a given safety property, counterfactual reasoning is used to distinguish component failures that actually contributed to the outcome from failures that had no impact on the violation of the property. Blaming was implemented in [29] through a reduction to a model checking problem for timed automata.

Debugging of LNT descriptions was also considered, in addition to the classical verification features of CADP. Salaün and Ye [30] devised a coverage analysis based on inserting *probes* (special actions) at suitable places in an LNT description without disturbing its behavior (i.e., the inserted probes, if considered as internal actions, yield a behavior branching bisimilar to the original one). The probes enable to track the execution of decisions and statement blocks in the underlying LTS model, and thus to detect lacks in coverage and/or anomalies in the LNT description.

Barbon *et al.* [31] proposed an approach to facilitate the analysis of (sequence) counterexamples produced by a model checker when evaluating a temporal property on an LNT description. This is achieved by spotting, in a given counterexample, the actions triggering a switch of the system execution from incorrect to correct behavior. These actions indicate possible causes of errors, being especially useful for large and intricate counterexamples. Both approaches [30, 31] have been automated in connection with CADP, but are generally applicable to formal languages with action-based, interleaving semantics.

[4] By construction, all these states are reachable.

6 Conclusion

In this paper, we introduced a method that allows to use models written in LNT for fault localization. The underlying methodology is based on the concept of wrapper components that are themselves written in LNT. There the idea is to introduce a variable representing the health state of the component and to distinguish the correct behavior implemented in the original LNT model from the faulty behavior where a simulator can use all domain values for the component's parameters. The approach is not limited to capture the unknown faulty behavior but also to introduce failure modes together with their corresponding models, e.g., in order to introduce stuck-at faults. The models are used together with a script to find all single faults of a system via setting one health variable for a component to faulty after the other and stating the rest of the components as working as expected.

Besides the underlying foundations, we also present a case study using the well-known data encryption standard (DES) were we are able to determine all the manually introduced faults using the proposed model. For model-based diagnosis the advantages are (1) to be able to make use of a modeling language that was developed for system verification for diagnosis, and (2) to obtain a rich set of already developed models and tools, which can now be further reused for fault localization.

Future research will include further improving diagnosis via more closely integrating the LNT tools with available diagnosis algorithms and further experiments to assess the potential and scalability of the approach. In particular, we plan to experiment with different failure models in LNT, multiple faults, and more available LNT models of other case studies.

Acknowledgments

We are grateful to Hermann Felbinger and Josip Bozic for their very much helpful discussions finally leading to this paper and the LNT model wrapper. The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 865248 (Securing Web Technologies with Combinatorial Interaction Testing - SecWIT), RIDINGS (RIgorous DesIgN of GALS Systems) project of the PHC Amadeus program, and the Région Auvergne-Rhône-Alpes within the program ARC 6.

References

[1] Yannick Pencolé and Marie-Odile Cordier. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artif. Intell.*, 164(1-2):121–170, 2005.

[2] Gianfranco Lamperti and Marina Zanella. Context-sensitive diagnosis of discrete-event systems. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 969–975. IJCAI/AAAI, 2011.

[3] Johan de Kleer, Bill Janssen, Daniel G. Bobrow, Tolga Kurtoglu, Bhaskar Saha, Nicholas R. Moore, and Saravan Sutharshana. Fault augmented Modelica models. In *24th Int. Workshop on Principles of Diagnosis (DX)*, pages 71–78, 2013.

[4] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. From LOTOS to LNT. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 3–26. Springer-Verlag, October 2017.

[5] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[6] Luca Console, Daniele Theseider Dupré, and Pietro Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[7] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[8] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnosis and systems. *Artificial Intelligence*, 56(2-3):197–222, 1992.

[9] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[10] Iulia Nica, Ingo Pill, Thomas Quaritsch, and Franz Wotawa. The route to success - a performance comparison of diagnosis algorithms. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1039–1045, Bejing, China, 2013.

[11] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe, and Gideon Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.7). INRIA, Grenoble, France, July 2017.

[12] ISO/IEC. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1989.

[13] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[14] Hubert Garavel and Wendelin Serwe. The Unheralded Value of the Multiway Rendezvous: Illustration with the Production Cell Benchmark. In Holger Hermanns and Peter Höfner, editors, *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems (MARS'17), Uppsala, Sweden*, volume 244 of *Electronic Proceedings in Theoretical Computer Science*, pages 230–270, April 2017.

[15] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 15(2):89–107, April 2013.

[16] Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In Jorge Cuellar, Tom Maibaum, and Kaisa

Sere, editors, *Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland*, volume 5014 of *LNCS*, pages 148–164. Springer Verlag, May 2008.

[17] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[18] Wendelin Serwe. Formal Specification and Verification of Fully Asynchronous Implementations of the Data Encryption Standard. In Rob van Glabbeek, Jan Friso Groote, and Peter Höfner, editors, *Proceedings of the International Workshop on Models for Formal Analysis of Real Systems (MARS'15), Suva, Fiji*, volume 196 of *Electronic Proceedings in Theoretical Computer Science*. Open Publishing Association, 2015.

[19] National Institute of Standards and Technology. Data encryption standard (DES). Federal Information Processing Standards Publication 46-3, October 1999.

[20] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[21] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993.

[22] Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis*. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada, 1994.

[23] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.

[24] Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*, pages 1074–1079, Stockholm, Sweden, August 1999.

[25] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling Java Programs for Diagnosis. In *ECAI*, pages 171–175, Berlin, Germany, 2000. IOS Press.

[26] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016.

[27] Ingo Pill and Thomas Quaritsch. Behavioral diagnosis of LTL specifications at operator level. In *International Joint Conference on Artificial Intelligence*, pages 1053–1059, 2013.

[28] Bernhard Peischl, Ingo Pill, and Franz Wotawa. Using Modelica programs for deriving propositional horn clause abduction problems. In *KI*, volume 9904 of *Lecture Notes in Computer Science*, pages 185–191. Springer, 2016.

[29] Gregor Gössler and Lăcrămioara Aştefănoaei. Blaming in component-based real-time systems. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT'14, pages 7:1–7:10, New York, NY, USA, 2014. ACM.

[30] Gwen Salaün and Lina Ye. Debugging Process Algebra Specifications. In *VMCAI 2015*, volume 8931, page 18, Mumbai, India, January 2015. Springer.

[31] Gianluca Barbon, Vincent Leroy, and Gwen Salaün. Debugging of Concurrent Systems Using Counterexample Analysis. In Mehdi Dastani and Marjan Sirjani, editors, *Revised selected papers of 7th International Conference on Fundamentals of Software Engineering (FSEN 2017), Tehran, Iran*, volume 10522 of *Lecture Notes in Computer Science*, pages 20–34. Springer-Verlag, April 2017.