Debugging Process Algebra Specifications

Gwen Salaün, Lina Ye

Debugging Process Algebra Specifications

Gwen Salaün[1] and Lina Ye[2]

[1] University of Grenoble Alpes, Inria, LIG, CNRS, France
[2] Department of Computer Science, Supélec, France

Abstract. Designing and developing distributed and concurrent applications has always been a tedious and error-prone task. In this context, formal techniques and tools are of great help in order to specify such concurrent systems and detect bugs in the corresponding models. In this paper, we propose a new framework for debugging value-passing process algebra through coverage analysis. We illustrate our approach with LNT, which is a recent specification language designed for formally modelling concurrent systems. We define several coverage notions before showing how to instrument the specification without affecting original behaviors. Our approach helps one to improve the quality of a dataset of examples used for validation purposes, but also to find ill-formed decisions, dead code, and other errors in the specification. We have implemented a tool for automating our debugging approach, and applied it to several real-world case studies in different application areas.

1 Introduction

Recent computing trends promote the development of software applications that are intrinsically parallel, distributed and concurrent. However, designing and developing distributed software has always been a tedious and error-prone task, and the ever increasing software complexity is making matters even worse. Therefore, it is impossible for any human being to foresee all the possible executions of this kind of application, which thus can hardly be free of bugs. In this context, formal techniques and tools are of great help in order to detect bugs in abstract models of concurrent systems. Although we are still far from proposing techniques and tools avoiding the existence of bugs in complex, real-world software systems, we know how to automatically chase and find bugs that would be very difficult, if not impossible, to detect manually.

A variety of formal specification languages has been developed over the last few decades, such as algebraic specifications (CASL), state-based formalisms (VDM, Z, B), automata-based languages (FSM, UML state diagrams, Statecharts), Petri nets or (value-passing) process algebras. Process algebras were designed for modelling concurrent systems and present several advantages compared to similar specification languages (such as automata-based languages or Petri nets): they are equipped with formal semantics, compositional notations, and are expressive enough to provide several levels of abstraction (*e.g.*, data with LOTOS or mobility with π-calculus); real-world systems can be specified

using textual notations, and there exist several verification toolboxes for them (CADP, mCRL2, LTSA, FDR2, etc.). In contrast, the syntax of process algebras is still hard to understand and use, particularly for non-experts. In order to fill this gap, LNT [7] was proposed a few years ago. LNT is a value-passing process algebra inspired from the E-LOTOS standard [19] and from imperative programming languages. LNT supports both the description of complex data types and of concurrent processes using the same user-friendly syntax. LNT specifications can be analyzed using CADP [13], a toolbox that provides various verification techniques and tools such as model checking, compositional verification, or performance evaluation. LNT is already used by several universities for teaching and research purposes, and by companies (such as STMicroelectronics or Orange labs) for designing and verifying different kinds of systems.

When using model checking techniques as those available in CADP, we usually have an LNT specification of a system, a dataset of validation examples, and a set of temporal properties to be verified on the system being designed. When we apply the LNT specification on a validation example, we obtain a Labelled Transition System (LTS), which corresponds to all the possible executions of the specification for this example. These LTSs are computed automatically using CADP exploration tools (enumerative approach). A validation example defines a set of inputs to the LNT specification and is similar to a test case in the testing domain. The LTS generation without explicit inputs might turn out to be impossible due to the enumeration on possibly infinite data domains. Bounding the exploration is a solution but this often results in huge LTSs (state explosion), which are therefore very long to analyze. This is why, in this setting, we prefer to work with a set of concrete inputs that we call validation examples in this paper.

The aforementioned properties can be verified on the generated LTS using model checking techniques. At this stage, building the set of validation examples and debugging the system is a real burden, in particular for non-experts. Counterexamples (sequences of actions violating the property) provided by model checkers are the only feedback one may have, and analysing such diagnostics may be very complicated, especially when the counterexample consists of hundreds of actions. More precisely here are a couple of issues that may arise during this phase: (i) we do not know whether the set of validation examples covers all the possible execution scenarios described in the LNT specification; (ii) the LNT specification may contain ill-formed decisions, non-synchronizable actions, and dead code, which require to be corrected, and are not necessarily found using model checking techniques.

Structural coverage is considered as one important metric of software quality and is normally used in implementation testing [21]. Coverage criteria can guide the selection of test cases as well as software reliability estimation. One common approach is to use coverage analysis for measuring the quality of the suite of test cases, which is often evaluated by its ability to detect mutants, *i.e.,* potential faults that are artificially inserted [16]. Several coverage criteria are well established, such as instruction coverage, decision coverage, data-flow coverage,

and path coverage. In this paper, we explore a different angle of the same question that relates to specification coverage. We demonstrate how to improve the quality of validation examples, and more importantly to debug specifications through coverage analysis. Formal specification languages have already benefited from tool-supported coverage metrics, such as SDL with Telelogic's Tau that measures the coverage of states and transitions, and VDM with IFAD's VDMTools [2].

In this paper, we are interested in debugging value-passing process algebra through coverage analysis, and we applied it to LNT specifications. We first define block, decision, and action coverage for specifications before showing how to insert probes to collect coverage information. Then we present how to analyze coverage based on the collected information in two steps. In the first step, we simultaneously analyze block and decision coverage to locate uncovered areas. We define a relationship between blocks and decisions, which is used to detect ill-formed decisions as well as to choose the uncovered parts that may contain non-synchronizable actions. In the second step, we perform action coverage analysis in these selected uncovered parts to find out the non-synchronizable actions. We implemented a tool to automate our approach, and we applied it to more than one hundred LNT specifications including six real-world case studies. It is worth emphasizing that we found several important issues for these specifications (*e.g.,* incomplete dataset of validation examples, ill-formed decisions, non-synchronizable actions, and dead code).

The main contributions of this paper are as follows:

- We developed new techniques to debug formal specifications, illustrated by LNT.
- We proved that applying our techniques has no impact on the original behaviors of the system by proving branching equivalence preservation.
- We implemented these techniques as a tool, CAL, built on top of the publicly available and widely-used CADP verification toolbox.
- We applied CAL to more than one hundred LNT specifications including six real-world systems.

The rest of this paper is organized as follows. In Section 2, we briefly introduce LNT. In Section 3, our solution for LNT coverage analysis is presented, including how to insert probes without impact on the original system behaviors as well as how to compute coverage in two steps. Section 4 describes our implementation and experimental results. Sections 5 and 6 present related work and concluding remarks, respectively.

2 Overview of LNT

The LNT specification language is an improved variant of the E-LOTOS standard [19]. LNT combines the best features of imperative and functional programming languages on the one hand, and value-passing process algebras on the other. Therefore, LNT supports both the description of complex data types and of concurrent processes using the same user-friendly syntax. LNT formal operational semantics is defined in terms of LTSs. For the sake of brievity, we show in Table 1 the syntax and semantics of a fragment of LNT, where x_i and T_i represent a variable and its type respectively, E denotes a logical expression, V is either a variable or an expression with type coercion, and V_i are its possible values [7].

Table 1. Syntax and operational semantics of LNT fragment

$$\mathbb{B} ::= \textbf{stop} \mid \mathbb{B}_1;\ \mathbb{B}_2 \mid \textbf{select}\ \ \mathbb{B}_1 [] ... [] \mathbb{B}_n\ \textbf{end select}$$
$$\mid \textbf{par}\ G\ in\ \mathbb{B}_1 || ... || \mathbb{B}_n\ \textbf{end par} \mid \textbf{if}\ E\ \textbf{then}\ \mathbb{B}\ \textbf{end if}$$
$$\mid \textbf{case}\ V\ \textbf{in}\ V_1 \rightarrow \mathbb{B}_1 \mid ... \mid V_m \rightarrow \mathbb{B}_m\ \textbf{end case} \mid \textbf{while}\ E\ \textbf{loop}\ \mathbb{B}\ \textbf{end loop}$$

$$(SEQ1)\ \frac{\mathbb{B}_1 \xrightarrow{\beta} \mathbb{B}_1\prime}{\mathbb{B}_1; \mathbb{B}_2 \xrightarrow{\beta} \mathbb{B}_1\prime; \mathbb{B}_2} \qquad (SEQ2)\ \frac{\mathbb{B}_1 \xrightarrow{\delta} \mathbb{B}_1\prime \quad \mathbb{B}_2 \xrightarrow{\beta} \mathbb{B}_2\prime}{\mathbb{B}_1; \mathbb{B}_2 \xrightarrow{\beta} \mathbb{B}_2\prime}$$

$$(SEL)\ \frac{k \in [1,n] \quad \mathbb{B}_k \xrightarrow{\beta} \mathbb{B}_k\prime}{\textbf{select}\ \ \mathbb{B}_1 [] ... [] \mathbb{B}_n\ \textbf{end select} \xrightarrow{\beta} \mathbb{B}_k\prime}$$

$$(PAR)\ \frac{k \in [1,n] \quad \mathbb{B}_k \xrightarrow{\beta} \mathbb{B}_k\prime \quad gate(\beta) \neq G}{\textbf{par}\ G\ \textbf{in}\ \mathbb{B}_1 || ... || \mathbb{B}_n\ \textbf{end par} \xrightarrow{\beta} \textbf{par}\ G\ \textbf{in}\ \mathbb{B}_1 || ... || \mathbb{B}_k\prime || ... || \mathbb{B}_n\ \textbf{end par}}$$

$$(COM)\ \frac{I \subseteq [1,n] \quad \forall k \in I . \mathbb{B}_k \xrightarrow{\beta} \mathbb{B}_k\prime \quad gate(\beta) = G \qquad j \in I}{\textbf{par}\ G\ \textbf{in}\ \mathbb{B}_1 || ... || \mathbb{B}_n\ \textbf{end par} \xrightarrow{\beta} \textbf{par}\ G\ \textbf{in}\ \mathbb{B}_1 || ... || \mathbb{B}_j\prime || ... || \mathbb{B}_n\ \textbf{end par}}$$

$$(IF1)\ \frac{[\![E]\!] = true \quad \mathbb{B} \xrightarrow{\beta} \mathbb{B}\prime}{\textbf{if}\ E\ \textbf{then}\ \mathbb{B}\ \textbf{end if} \xrightarrow{\beta} \mathbb{B}\prime} \qquad (IF2)\ \frac{[\![E]\!] = false}{\textbf{if}\ E\ \textbf{then}\ \mathbb{B}\ \textbf{end if} \xrightarrow{\delta} stop}$$

$$(WHILE1)\ \frac{[\![E]\!] = true \quad \mathbb{B} \xrightarrow{\pi} \mathbb{B}\prime \quad \mathbb{B}\prime \xrightarrow{\delta} \mathbb{B}}{\textbf{while}\ E\ \textbf{loop}\ \mathbb{B}\ \textbf{end loop} \xrightarrow{\pi} \textbf{while}\ E\ \textbf{loop}\ \mathbb{B}\ \textbf{end loop}}$$

$$(WHILE2)\ \frac{[\![E]\!] = false}{\textbf{while}\ E\ \textbf{loop}\ \mathbb{B}\ \textbf{end loop} \xrightarrow{\delta} stop}$$

$$(CASE)\ \frac{j \in 1, ...m\ (\forall k \in 1, ..., j-1)[\![V == V_k]\!] = false, [\![V == V_j]\!] = true \quad \mathbb{B}_j \xrightarrow{\beta} \mathbb{B}\prime}{\textbf{case}\ V\ \textbf{in}\ V_1 \rightarrow \mathbb{B}_1 \mid ... \mid V_m \rightarrow \mathbb{B}_m\ \textbf{end case} \xrightarrow{\beta} \mathbb{B}\prime}$$

LNT processes are built from action, sequential composition (;), choice (**select**), parallel composition (**par**), condition (**if, case, while**), and termination (**stop**). Communication is carried out by rendezvous on gates G (multiple synchronization points) with bidirectional transmission of multiple values. For simplicity, in Table 2, we consider actions with only two values being sent in both directions. The gate on which an action β takes place is denoted by $gate(\beta)$, and we use π to denote a sequence of actions. Particularly, an action can be an emission (**!**) or a reception (**?**). The special action δ is used for successful termination. The internal action is denoted by the special gate i, which cannot be used for

synchronization. Processes are parameterized by sets of actions (alphabets) and input/output data variables.

LNT specifications can be analyzed using CADP [13], a verification toolbox dedicated to the design, analysis, and verification of asynchronous systems consisting of concurrent processes interacting via message passing.

3 Coverage Analysis

In this section, we show how to analyze structural coverage for LNT specifications, which helps one to improve the quality of the dataset of validation examples as well as to detect several issues in the specification, *i.e.,* ill-formed or unnecessary decisions, non-synchronizable actions, and dead code.

3.1 Terminology

One well-known coverage criteria is the instruction coverage, *i.e.*, the number of executed instructions out of the total number of instructions. It is used for measuring code quality, *i.e.,* checking the existence of non-executed code. However, this coverage requires checking each instruction separately, which is not efficient for large programs. Since several instructions can be in the same block, for efficiency reasons it makes more sense to keep track of blocks rather than individual instructions. Note that 100% block coverage implies 100% instruction coverage. This is why we choose *block coverage* as the first criterion. However, from block coverage, we cannot deduce outcomes of decisions, *e.g.*, whether a loop reaches its termination condition or whether the false outcome of a decision is evaluated. To solve this, we consider *decision coverage* as the second criterion, which takes a more in-depth view of the program. Furthermore, note that for LNT, synchronization points between processes are modelled by rendezvous on synchronized actions. To check whether all actions are well designed to be synchronizable, we choose *action coverage* as a third criterion. It is a special metric for concurrent languages.

Let us define the notion of blocks for LNT. We first define control instructions in LNT that will be used to determine blocks.

Definition 1 *(LNT Control Instruction). The Control Instructions (CIs) of an LNT specification include conditional instructions (**if, case, while**), parallel and choice ones (**par, select**), and termination (**stop**).*

Definition 2 *(LNT Block). Given an LNT specification, an LNT block is the largest sequence of instructions free of CIs. Particularly, we call a block without action a silent block.*

Now we formally define the notion of coverage for blocks, actions, and decisions. In the following, we simply call LNT block as block if there is no ambiguity. In LNT, a decision is a Boolean expression composed of conditions and zero or more Boolean operators. Particularly, for case statements, each branch is considered as one decision. For example, given the following case statement:

$$\textbf{case } V \textbf{ in } \quad V_0 \rightarrow I_0 \ \mid V_1 \rightarrow I_1 \ \mid V_2 \rightarrow I_2 \ \textbf{ end case}$$

we have the following three decisions, one per branch:
- $V == V_0$;
- $V == V_1$;
- $V == V_2$.

Definition 3 *(Covered block, action, and decision). Let s be an LNT specification and ds be a dataset of validation examples. We have the following notions:*
- *a block b (an action a, resp.) in s is said to be covered w.r.t. ds if b $(a, resp.)$ is executed by at least one example $e \in ds$, denoted by $C^{s:B}_{ds}(b)$ $(C^{s:A}_{ds}(a), resp.)$, simply $C^B(b)$ $(C^A(a), resp.)$ if there is no ambiguity;*
- *a decision d in s is said to be covered w.r.t ds if $\exists e_1, e_2 \in ds$, such that the true outcome of d is evaluated by e_1 and the false outcome is evaluated by e_2, denoted by $C^{s:D}_{ds}(d)$, simply $C^D(d)$. Specially, if only true (false, resp.) outcome of d is evaluated, we denote this by $C^{D:t}(d)$ $(C^{D:f}(d), resp.)$.*

Definition 4 *(Block (Decision, Action, resp.) coverage). Let s be an LNT specification and ds be a dataset of validation examples. Block (Decision, Action, resp.) coverage w.r.t. s and ds, denoted by BC^s_{ds} (DC^s_{ds}, AC^s_{ds}, resp.), is the percentage of the number of covered blocks (decisions, actions, resp.) out of their total number. Formally, $BC^s_{ds} = \|B_c\|/\|B\|$ ($DC^s_{ds} = \|D_c\|/\|D\|$, $AC^s_{ds} = \|A_c\|/\|A\|$, resp.), where $B_c = \{b \in B \mid C^B(b)\}$ ($D_c = \{d \in D \mid C^D(d)\}$, $A_c = \{a \in A \mid C^A(a)\}$, resp.) and B (D, A, resp.) is the set of all blocks (decisions, actions, resp.) in the given specification. If there is no ambiguity, we simply denote the three coverage as BC, DC, and AC.*

3.2 Probe Insertion

To measure structural coverage of LNT, we instrument the code with probes in order to collect coverage information. Before showing how to do this, we first define LTS, which will be used to explicitly capture such coverage information.

Definition 5 *(LTS) An LTS is a tuple $L = (S_L, s^0_L, \Sigma_L, T_L)$ where S_L is a finite set of states; $s^0_L \in S_L$ is the initial state; Σ_L is a finite set of actions; $T_L \subseteq S_L \times \Sigma_L \times S_L$ is a finite set of transitions.*

Given an LTS obtained from applying an LNT specification on one validation example, the only elements of the specification contained in this LTS are actions. Hence, to analyze the structural coverage, we propose to insert probes as new actions, whose presence in the LTS explicitly shows their coverage information. When inserting such probes, it is important to preserve the original system behaviors when all probes are hidden as internal actions. It is reasonable to consider probes as internal actions because they are represented by fresh and non-synchronized actions, which do not interfere with the existing instructions. In the following, we denote the set of LTSs corresponding to the dataset of validation examples by Δ.

Block. To measure the block coverage, we insert a probe P at the end of each block. The presence of P in Δ implies that its associated block is covered, *i.e.,* $\exists L \in \Delta$, such that $P \in \Sigma_L$.

Decision. Table 2 illustrates how probes are inserted for decisions in LNT. For decision coverage, to obtain the evaluated outcome(s) of a given decision E, we equip the corresponding probe with this decision as its parameter, *i.e.*, P ($!E$). The parameter $!E$ displays the outcome of the decision E. Precisely, if the decision E is evaluated to both true and false for a validation example, then in its corresponding LTS, we have the action $P\ !TRUE$ as well as $P\ !FALSE$. Otherwise, if it is evaluated to only true (false, resp.), what we obtain in the LTS is $P\ !TRUE$ ($P\ !FALSE$, resp.). The decision E is covered if $\exists L_1, L_2 \in \Delta$, such that $P\ !TRUE \in \Sigma_{L_1}, P\ !FALSE \in \Sigma_{L_2}$.

In Table 2, for the **if** construct, we add the corresponding probe just before it to catch its outcome. For the **case** construct, its operational semantics is to sequentially pick the first condition that holds true. To capture such semantics, we first represent a decision for each branch by a different probe, *i.e.,* P_1 for $V = V_1$ and P_2 for $V = V_2$. At the beginning of each corresponding branch, we add its probe with parameter $TRUE$ and the probes representing all its precedent branches with parameter $FALSE$. In this way, only probes with evaluated decisions appear in the corresponding LTSs. For the loop construct (**while**), probes should be inserted both before and after the corresponding construct to guarantee that both outcomes of the decision are obtained if it is covered. Otherwise, with the probe only before the construct, we will never capture the false outcome if the value of decision is first true and then becomes false. With the probe only after the construct, the true outcome cannot be caught in the same situation.

Table 2. Probe insertion for decisions

Types	Before Insertion	After Insertion
If	**if** E **then** B_1 **end if**	$\underline{P(!E)}$; **if** E **then** B_1 **end if**
Case	**case** V **in** $V_1 \rightarrow B_1$ $\mid V_2 \rightarrow B_2$ **end case**	**case** V **in** $V_1 \rightarrow \underline{P_1(!TRUE)}; B_1$ $\mid V_2 \rightarrow \underline{P_1(!FALSE)}; \underline{P_2(!TRUE)}; B_2$ **end case**
While	**while** E **loop** B_1 **end loop**	$\underline{P(!E)}$; **while** E **loop** B_1 **end loop**; $\underline{P(!E)}$

Action. For action coverage, we insert a probe just after the target action, whose presence in an LTS indicates that this action is covered. Even though actions can be manifested by themselves in LTSs, probes are still necessary. The reason is that in an LNT specification, one action may be used several times at different places. Each appearance of an action is called its instance. The presence of an action in Δ does not mean that all its instances are covered. To determine which exact instance of an action is not yet covered if there is any, we use different probes to distinguish all action instances.

Critical block and decision. Now we define critical blocks and decisions that are located at the beginning of a choice branch, whose corresponding probes should be inserted in a different way to preserve system behaviors.

Definition 6 *(Critical block (decision)). Given a silent block (decision), if it is a subpart of a select construct such that there is no action before it in the corresponding choice branch, then it is called a critical block (decision).*

Intuitively, given a critical block or decision, if we insert its probe as described before, this probe becomes the first action in the corresponding choice branch. In this case, the branching structure will be altered ($\tau.a+b$ and $a+b$, in a CCS-like notation [20], are not branching equivalent). To solve this problem, Table 3 shows how to insert probes for critical blocks and decisions in a different way to keep the original behaviors, where B_i^s denotes a silent block. For a critical block, an additional variable, initialized as 0, is used to indicate whether this block is completely executed. This variable is then used as the parameter of the corresponding probe inserted after the choice construct. If the value is 1 (0, resp.), then this block is covered (not covered, resp.), represented by $P(!1)$ ($P(!0)$, resp.) in the corresponding LTS. For a critical decision, an extra variable, initialized as 2, is used as the parameter of the corresponding probe inserted after the choice construct. The value being 1 (0, resp.) represents true (false, resp.) outcome of the decision. Particularly, if the value is 2, then the decision is not even evaluated.

Table 3. Probe insertion for critical blocks and critical decisions

Criterion	Types	Before Insertion	After Insertion
Block		**select** B_1^s [] B_2 **end select**	tag:=0; **select** B_1^s; tag:=1 [] B_2 **end select** $P(!tag)$
Decision	If	**select** B_1^s; **if** E **then** B_2 **end if**; B_3 [] B_4 **end select**	tag:=2; **select** B_1^s; **if** E **then** tag:=1 **else** tag:=0 **end if**; **if** E **then** B_2 **end if**; B_3 [] B_4 **end select** $P(!tag)$
	Case	**select** B_1^s; **case** V **in** $V_1 \rightarrow B_2$ $\mid V_2 \rightarrow B_3$ **end case**; B_4 [] B_5 **end select**	tag1:=2; tag2:=2; **select** B_1^s; **case** V **in** $V_1 \rightarrow tag1 := 1; B_2$ $\mid V_2 \rightarrow tag1 := 0; tag2 := 1; B_3$ **end case**; ...**end select**; $P_1(!tag1);\ P_2(!tag2)$
	While	**select** B_1^s; **while** E **loop** B_2 **end loop**; $B_3[]B_4$ **end select**	tag1:=2; tag2:=2; **select** B_1^s; **if** E **then** tag1:=1 **else** tag1:=0 **end if**; **while** ... **end loop**; **if** E **then** tag2:=1 **else** tag2:=0 **end if**; $B_3[]B_4$ **end select**; $P(!tag1);\ P(!tag2)$

3.3 Behavior Preservation

In this section, we prove the behavioral equivalence between the original LNT specification and the one with inserted probes hidden as internal actions, which

is called an extended specification in the following. We consider here branching bisimulation, which is one of the finest behavioral equivalences studied in process theory [22]. This equivalence preserves the branching structure of systems by considering all intermediate states including those with internal transitions. We prove the branching equivalence directly on LNT specification, which is actually a process algebra. The underlying model of process algebra is its corresponding LTS, where each process represents a state in its LTS.

Definition 7 *(Branching bisimulation). A branching bisimulation relation R is a binary relation over a process algebra such that it is symmetric and satisfies the following transfer property: if pRq and $p \xrightarrow{a} p'$, then one of the two following conditions should be satisfied:*
- *$a = \tau$ and $p'Rq$;*
- *there is a sequence of transitions $q \xrightarrow{\tau^*} q'' \xrightarrow{a} q'$, pRq'' and $p'Rq'$.*

If there is a branching bisimulation relation R between p and q, then p and q are branching bisimilar, denoted by $p \approx_b q$.

Theorem 1 *Let s be an LNT specification, s' be its corresponding extended specification (both s and s' are processes), then s and s' are branching bisimilar, i.e., $s \approx_b s'$.*

Proof. From Definition 7 and the fact that the only difference between s and s' is the set of inserted probes that are considered as internal actions, it follows that to prove this theorem, we have to show that $\forall \tau_P \in s'$, where τ_P represents a probe considered as an internal action, for a binary relation R, the condition Υ is satisfied, where Υ: $\forall \tau_P \in s', p \xrightarrow{\tau_P} p' \Rightarrow pRp'$. This means that any inserted probe has no impact on the original behaviors in terms of branching structures. Next we demonstrate, without loss of generality, that this is true for each probe.

1. For an action a in any composition or construct, it can be directly deduced that its corresponding probe satisfies the condition Υ, from the silent step law in process algebra, denoted by L_τ: $a.\tau_P \approx_b a$ (CCS-like notation, which will be used in the following for the sake of brievity).
2. For a block B, we analyze its corresponding probe in three different constructs separately, *i.e.,* sequential, parallel and choice.
   - For B in a sequential composition that is not inside any parallel and choice construct, its probe satisfies the condition Υ since it is not possible for this probe to change the branching structure.
   - For B in a parallel composition that is not inside any choice construct, there are two possible situations. One is that the corresponding probe τ_P inserted for B is the first action in the corresponding parallel branch, where B must be a silent block. Another one is that τ_P is not the first action in this branch. For the latter one, τ_P satisfies Υ from L_τ. Now we analyze the first situation in the following way.
     Base case: consider $(\tau_P.a)||b$, for which we have $\tau_P.a||b = \tau_P.(a||b) + b.\tau_P.a$. From this, the τ_P in $b.\tau_P.a$ satisfies Υ. Moreover, from L_τ, we further get $\tau_P.(a||b) + b.\tau_P.a = \tau_P.(a||b) + b.a$. Now we show that this

τ_P also satisfies Υ because $b.a$ is included in $a||b$ since $a||b = a.b + b.a$.
Induction: now consider $(\tau_P.a)|||(b_1.....b_n) = \tau_P.(a||(b_1.....b_n))+b_1.((\tau_P.a)|||(b_2.....b_n))$. Now we suppose that τ_P in $(\tau_P.a)|||(b_2.....b_n)$ satisfies Υ and thus can be reduced to $a||(b_2.....b_n)$. This follows that the first τ_P also satisfies Υ since $b_1.(a||(b_2.....b_n))$ is included in $(a||(b_1.....b_n))$. In the same way, the induction is applied to structures with more than two parallel branches.

- For B in a parallel composition that itself is inside a choice construct, suppose that there is no action before B in the corresponding branch and B is a silent block. Then B is a critical block and its corresponding probe is inserted after the choice construct, which satisfies Υ.
- For B in a choice construct, either B is a critical block, or there is an action before B. For the former case, the corresponding probe is inserted after the choice construct and thus satisfies Υ. The probe of the latter case also satisfies Υ from L_τ.

3. For a decision E, if it is not critical, the probe satisfy Υ from L_τ. If it is critical, we avoid $\tau_P.a + b \not\approx_b a + b$ by inserting the corresponding probe in the sequential composition after the choice construct, which then satisfies Υ. Actually the demonstration follows the exact same line as described above for blocks.

Now we have shown that each probe inserted as described in Section 3.2 does not alter the original behaviors of the system in terms of branching structure and this proves this theorem. ■

3.4 Coverage Computing

If we simultaneously insert probes for all three criteria to compute their coverage, the corresponding LTSs would suffer from the state explosion problem. To solve this, we separate the coverage analysis into two steps. In a first step, we insert probes for blocks and decisions to reveal those uncovered. The entry of a block may be controlled by the outcome of a decision, *e.g.*, the true outcome of an **if** instruction allows the execution to enter its associated block. For such a block, its coverage may be prevented by two possible reasons: the outcome of its controlling decision prohibits the execution from entering it, or only a part of the block is executed due to non-synchronizable actions. In a second step, we are more interested in those partially covered blocks whose entry is allowed by a decision to discover non-synchronizable actions.

Definition 8 *(Dependency of block on decision).*

- *Given a block b and a decision d, if the execution of b is dependent of the true (false, resp.) outcome of d, this dependency is denoted by* $b \Rightarrow_{pd} d$ *(*$b \Rightarrow_{nd} d$*, resp.).*
- *If* $b \Rightarrow_{pd} d$ *or* $b \Rightarrow_{nd} d$*, we denote it* $b \Rightarrow_d d$*.*

A block whose entry is allowed is an executable block. Such a block either has no dependent decision or is permitted to be entered by its associated decision.

In other words, if an executable block is dependent of the true (false, resp.) outcome of a decision d, then this outcome of d is covered.

Definition 9 *(Executable block). A block b is executable if one of the following conditions is satisfied:*
- $\nexists d$ *such that* $b \Rightarrow_d d$;
- *if* $b \Rightarrow_d d$, *then either* $b \Rightarrow_{pd} d$ *and* $C^{D:t}(d)$, *or* $b \Rightarrow_{nd} d$ *and* $C^{D:f}(d)$.

Definition 10 *(Partially covered block). A block is a partially covered block if it is executable but not covered.*

Figure 1 overviews our coverage analysis in two steps. In the first step, we repeatedly apply the specification with probes for both blocks and decisions on each validation example to obtain the corresponding LTS. Block and decision coverages are simultaneously analyzed on these LTSs to obtain their coverage results, denoted by R_{BC} and R_{DC}, respectively. We have $R_{BC} = \{BC, \Gamma_{UB}\}$ and $R_{DC} = \{DC, \Gamma_{C^{D:t}}, \Gamma_{C^{D:f}}\}$, where BC (DC, resp.) is the percentage of block (decision, resp.) coverage, Γ_{UB} is the set of uncovered blocks, and $\Gamma_{C^{D:t}}$ ($\Gamma_{C^{D:f}}$, resp.) is the set of decisions whose true (false, resp.) outcome is covered. We can deduce whether an uncovered block is executable and thus calculate the set of partially covered blocks with R_{BC} and R_{DC}. In the second step, we insert probes for actions in this set of blocks before obtaining the corresponding LTSs and then perform action coverage analysis. The result of action coverage is $R_{AC} = \{AC, \Gamma_{PA}\}$, where AC is the percentage of action coverage, and Γ_{PA} is the set of non-synchronizable actions.

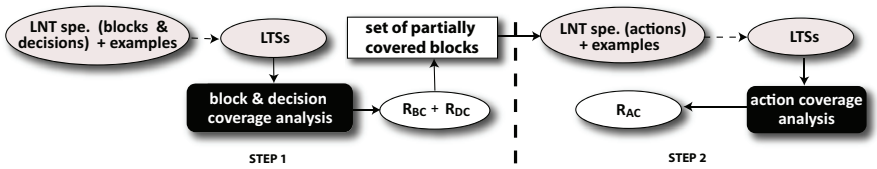

Fig. 1. Overview of coverage analysis in two steps.

3.5 Results Analysis

Given the coverage results described in the precedent section, two reasons can explain why the coverage percentages are lower than 100%:
1. lack of validation examples;
2. defects contained in the corresponding LNT specification.

For the first reason, the solution is to add examples that can explore those missing execution scenarios. For instance, suppose that the false outcome of a decision is never covered by the current dataset, we should add examples where

the value of this decision can be evaluated to false. If there is no such examples, then we should consider the second reason. For example, if we define one specification with two input parameters and both have two possible values, then we have in total four validation examples. In this case, if a coverage percentage cannot achieve 100%, then there must be some errors in the specification since there is no other possible examples (for illustration see the case study named AgtReconfig in Section 4). We list in the following the different types of errors that may be the source of the uncovered parts, which can be deduced thanks to the results obtained in the precedent section, *e.g.,* $\Gamma_{C^{D:f}}, \Gamma_{C^{D:t}}, \Gamma_{PA}$, etc.:

- **Ill-formed decision**: given a decision d such that $d \in \Gamma_{C^{D:f}}$ and $d \notin \Gamma_{C^{D:t}}$ ($d \in \Gamma_{C^{D:t}}$ and $d \notin \Gamma_{C^{D:f}}$, resp.), if $\exists b$, such that $b \Rightarrow_{pd} d$ ($b \Rightarrow_{nd} d$, resp.), this means that the uncovered outcome of a decision controls at least one block. Such situation is probably due to an ill-formed decision. For example, if a block is within an **if** conditional construct that always has false outcome, then this block is never covered.
- **Unnecessary decision**: given a decision d such that $d \in \Gamma_{C^{D:f}}$ and $d \notin \Gamma_{C^{D:t}}$ ($d \in \Gamma_{C^{D:t}}$ and $d \notin \Gamma_{C^{D:f}}$, resp.), if $\nexists b$, such that $b \Rightarrow_{pd} d$ ($b \Rightarrow_{nd} d$, resp.), this means that the uncovered outcome of a decision controls no block. Such decisions can be safely removed, *e.g.*, the false outcome of an **if** conditional construct is never achieved.
- **Non-synchronizable actions**: for an action a, if $a \in \Gamma_{PA}$, then its corresponding synchronization is ill-designed, *i.e.,* there is bad match between the received and the sent parameter types of the corresponding actions.
- **Dead code**: a piece of unreachable code in an uncovered block $b \in \Gamma_{UB}$ is called dead code if it is not due to the errors described above. This may be caused for example by wrong location of **stop**.

4 Evaluation

We have implemented our approach as a tool called CAL (Coverage Analysis of LNT). In this section, we first present the architecture of CAL joined with CADP before showing some experimental results. We also show how our two-step analysis can reduce the state space explosion problem compared to a more naive approach, where the three coverage criteria are simultaneously computed.

4.1 Implementation

The architecture of CAL with the cooperation of CADP is shown in Figure 2. The input of CAL is an LNT specification with a dataset of validation examples. The LNT specification is instrumented with probes for different criteria as described in Section 3.2. Then CAL calls CADP compilers to repeatedly apply the instrumented LNT specification on each validation example to obtain its corresponding explicit LTS. In this way, we can obtain a set of LTSs associated to the dataset of examples. Afterwards, the ANALYSER tool of CAL measures the coverage percentage and provides other results as described in Section 3.4.

All experiments were conducted on a server machine that has six 3.07 GHz processors and 11.7 GB of RAM. Considering that CADP has interfaces for reading LTSs that can be used by an application program written in C or C++, CAL is implemented in C, using gcc with version 3.4.3. The version of CADP used in our evaluation is BETA-VERSION 2014-c "Amsterdam".

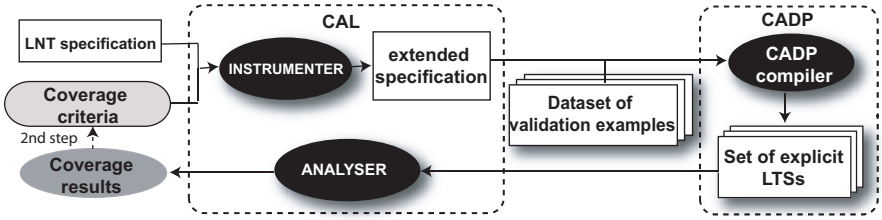

Fig. 2. Implementation architecture of CAL with CADP.

4.2 Experimental Results

To evaluate our approach, we have applied our tool to more than one hundred LNT specifications, including six real-world case studies in different application areas (hardware, cloud computing, multi-agent systems, and synchronization protocols). Table 4 lists the six case studies with their designer and a short description.

Table 5 lists both the size of the six case studies, *i.e.*, number of lines and validation examples, and their coverage results. Their size varies from 196 to 3700 lines. The number of validation examples differs from several to 200, which depends on the available input domain. For example, in the specification of AgtReconfig, the major process is defined with only two parameters that has two possible values. In this case, we can have four validation examples in total.

In this table, we show two versions for the Synchro case study, the first version is called Synchro1 and the second one Synchro2. The second version was obtained using our coverage results on the first version, as described in Section 3.4 and Section 3.5. Particularly, the block coverage was improved from 62.1% to 100%. This demonstrates the interest of the subsequent utilization of the measured coverage information computed by our approach. Precisely, to achieve 100% for this case study, the authors have not only added 12 complementary validation examples but have also corrected several non-synchronizable actions.

Another point is that for several case studies (DirectCache, DisCache, and ReConfig), all uncovered blocks were not executable, which is shown in Figure 3.

Table 4. Details of six real-world case studies.

Case Study	Designer	Description
DirectCache	STMicroelectronics	deals with cache coherence in multiprocessor systems by using a common directory.
AgtReconfig	Inria	provides an agent-based mechanism allowing distributed applications to be reconfigured at run-time [8].
DisCache	STMicroelectronics	ensures data consistency in multiprocessor shared memory systems that allow multiple copies of a datum [1].
SelfConfig	Inria, Orange labs	automates the configuration of a cloud application that is distributed on more than one virtual machine without requiring any centralized server [9, 24, 10].
ReConfig	Inria, Orange labs	reconfigures a running system composed of a set of interconnected components, where multiple failures occurring at reconfiguration time are tolerated [5].
Synchro	Inria	realizes the multiway rendezvous of LNT, where all parallel processes are organized in a hierarchical structure [11].

Table 5. Experimental results, where N_L: number of lines, N_{VE}: number of validation examples, N_B ($N_D, N_A, resp.$): number of blocks (decisions, actions, *resp.*), BC ($DC, AC, resp.$): block (decision, action, resp.) coverage.

	DirectCache	AgtReconfig	DisCache	SelfConfig	ReConfig	Synchro1	Synchro2
N_L	196	785	981	1635	3700	486	480
N_{VE}	5	4	6	60	200	18	30
N_B	12	31	33	31	90	66	66
BC	83.3%	67.7%	93.9%	83.8%	97.8%	62.1%	100%
N_D	12	27	23	23	89	50	50
DC	83.3%	74.1%	91.3%	73.9%	92.1%	60%	100%
N_A	9	50	33	32	53	72	72
AC	100%	64%	100%	93.8%	96.2%	68.1%	100%

In this case, the first step of coverage analysis is sufficient. For other case studies, the majority of uncovered blocks were not executable. This means that we consider very few blocks in the action analysis.

Besides improving the quality of validation examples, our coverage analysis also identified all types of errors described in Section 3.5. For example, several crucial ill-formed decisions affecting the whole system behaviors were detected and corrected for ReConfig, which were not discovered by model checking. Another point that we want to emphasize is that the analysis results can help in correcting the corresponding bugs in the implementation. In ReConfig for instance, the ill-formed decisions in the LNT specification, detected in the LTS

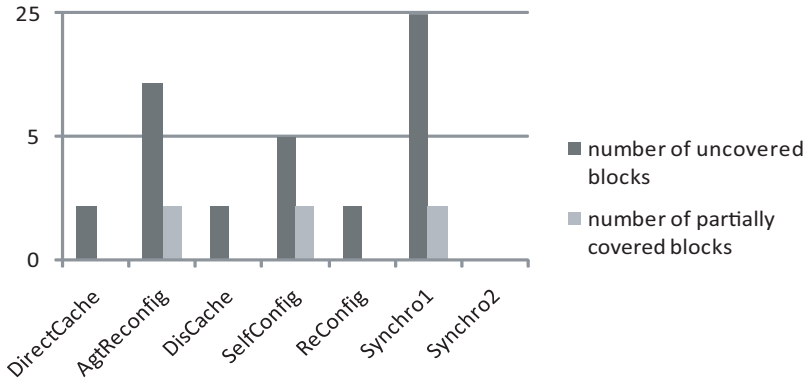

Fig. 3. Executability of uncovered blocks.

models, helped the developers of the corresponding Java implementation (Orange labs) to locate and correct them immediately.

To show the efficiency of our approach, we compare it with a more naive approach, where three criteria are simultaneously analyzed. We greatly reduce the number of probes for all case studies that we tested when adopting our approach in two steps. Take AgtReconfig as example, the total number of probes is 108 with the naive approach and is only 60 with ours. In our experiments, the reduced number of states and transitions for all case studies are between 30% and 60% thanks to the reduced number of probes. Furthermore, for some validation examples of Synchro2, we could not even construct the corresponding LTSs using the naive approach within a reasonable time (a few hours) but succeeded using ours within one hour.

5 Related Work

Step-by-step execution for LOTOS is proposed in [17], which is also called interactive simulation. The authors take the role of the environment by providing events to the specification and then by observing the results. Although useful for debugging, step-by-step execution is probably the simplest and weakest validation technique available for LOTOS. In [4], the authors propose to measure the completeness of an example suite in terms of the structural coverage described in LOTOS, where a probe is inserted after every action to check its achievement. However, they do not consider the decisions, whose coverage may have an important impact on the action coverage. The authors of [12] consider action, decision, and condition coverage for LOTOS. These criteria are measured totally separately. Furthermore, both works do not check behavioral equivalence between the original specification and the extended one. By using new actions as probes,

their insertion techniques imply that only weak trace equivalence is preserved. This is the weakest equivalence and thus not suitable for safety-critical systems, where altering branching structure could have serious consequences since an internal transition may alter the desired behavior of the system. Furthermore, compared to keeping weak trace equivalence, we guarantee the finest branching equivalence with probes considered as internal action, which however does not degrade the performance. The reason is that to preserve branching equivalence, as described in Section 3.2, for each critical block and critical decision, we only move their corresponding probe from inside the corresponding choice construct to after it. In other words, the number of probes required is not increased.

In [18], the authors propose an approach to test specifications by first formulating properties that should hold in the specification and then applying model checking or theorem proving to find violations. However, it is very difficult to select the set of properties such that they can evaluate all behaviors in the specification. This is also the case for LNT specification, where some faults detected by our coverage analysis cannot be identified by model checking. Model checking techniques are also used to automatically generate test cases that satisfy coverage criteria [15]. Similarly, in [14], a suite of test sequences are generated from SCR requirements specification by using a model checker's ability to construct counterexamples. Differently, our approach does not only improve the quality of validation examples, but more importantly detect faults in the specification through coverage analysis.

Coverage based testing is a widely used technique in software engineering and different coverage criteria are described in classical books on software testing, *e.g.*, [21]. Test coverage is considered as an essential factor to enhance new proposed models for software reliability estimation. For example, Piwowarsky *et al.* [23] predict software reliability based on the fact that the fault removal rate is a linear function of the code coverage. Cai and Lyu [6] propose to incorporate testing time and test coverage together into one single mathematical form to estimate the software reliability. However, in this paper, our goal is not to discover the quantitative relation between coverage analysis and fault detection rate but to directly debug formal specification by using coverage techniques.

6 Conclusion

In this paper, we have proposed a new approach to debug process algebra specifications, illustrated by LNT. First, we have introduced several coverage notions before showing how to insert probes to measure them by keeping the same behaviors. Second, we have proposed the coverage analysis in two steps such that we are able to find out uncovered parts keeping the number of probes as small as possible. The obtained results can be considered as efficient guides to either complete validation examples or correct errors in the given specification. Third, we have applied our implemented tool, CAL, to six real-world case studies. It is worth pointing out that our approach can also be applied to other value-passing process algebra such as CSP with FDR2 or Promela with SPIN.

So far we have defined an elementary set of coverage criteria, therefore one perspective of our work is to extend to other criteria for coverage analysis, such as multiple condition coverage, modified condition/decision coverage variants, or some criteria based on data flow [3].

Acknowledgements This work has been supported by the OpenCloudware project (2012-2015), which is funded by the French *Fonds national pour la Société Numérique* (FSN), and is supported by *Pôles* Minalogic, Systematic, and SCS. We would like to thank Radu Mateescu for his valuable suggestions to improve the paper.

References

1. Y. Afek, G. Brown, and M. Meritt. Lazy Caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, 1993.
2. S. Agerholm and P. G. Larsen. The IFAD VDM Tools: Lightweight Formal Methods. In *Proc. of FM-Trends'98*, volume 1641 of *LNCS*, pages 326–329. Springer, 1998.
3. P. Ammann, J. Offutt, and W. Xu. *Coverage Criteria for State Based Specifications, Formal Methods and Testing*. Springer, 2008.
4. D. Amyot and L. Logrippo. Structural Coverage for LOTOS - a Probe Insertion Technique. In *Proc. of TestCom'00*, pages 19–34. Kluwer, B. V., 2000.
5. F. Boyer, O. Gruber, and G. Salaün. Specifying and Verifying the SYNERGY Reconfiguration Protocol with LOTOS NT and CADP. In *Proc. of FM'11*, volume 6664 of *LNCS*, pages 103–117. Springer, 2011.
6. X. Cai and M. R. Lyu. Software Reliability Modeling with Test Coverage: Experimentation and Measurement with a Fault-Tolerant Software Project. In *Proc. of ISSRE'07*, pages 17–26. IEEE, 2007.
7. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator, Version 5.4. INRIA/VASY, 2011.
8. M. A. Cornejo, H. Garavel, R. Mateescu, and N. D. Palma. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. In *Proc. of DAIS'02*, volume 70 of *IFIP AICT*, pages 229 – 242. Springer, 2002.
9. X. Etchevers, T. Coupaye, F. Boyer, N. De Palma, and G. Salaün. Automated Configuration of Legacy Applications in the Cloud. In *Proc. of UCC'11*, pages 170–177. IEEE Computer Society, 2011.
10. X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. De Palma. Reliable Self-deployment of Cloud Applications. In *Proc. of SAC'14*, pages 1331–1338. ACM, 2014.
11. H. Evrard and F. Lang. Formal Verification of Distributed Branching Multiway Synchronization Protocols. In *Proc. of FORTE'13*, volume 7892 of *LNCS*, pages 146–160. Springer, 2013.
12. G. Fraser, M. Weiglhofer, and F. Wotawa. Coverage Based Testing with Test Purposes. In *Proc. of QSIC'08*, pages 199–208. IEEE, 2008.
13. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of TACAS'11*, volume 6605 of *LNCS*, pages 372–387. Springer, 2011.

14. A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Proc. of ESEC/FSE'99*, volume 24, pages 146–162. ACM, 1999.
15. A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, 2001.
16. R. Gopinath, C. Jensen, and A. Groce. Code Coverage for Suite Evaluation by Developers. In *Proc. of ICSE'14*. ACM, 2014.
17. R. Guillemot and L. Logrippo. Derivation of Useful Execution Trees from LOTOS Specifications by using an Interpreter. In *Proc. of FORTE'88*, pages 311–325. North-Holland Publishing Co., 1988.
18. C. L. Heitmeyer, J. Kirby, B. G. Labaw, M. Archer, and R. Bharadwaj. Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications. *IEEE Trans. Software Eng.*, 24(11):927–948, 1998.
19. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437 : 2001, International Organization for Standardization — Information Technology.
20. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
21. G. J. Myers. *The Art of Software Testing, Second Edition*. John Wiley & Sons, Inc., 2004.
22. R. De Nicola and F. Vaandrager. Three Logics for Branching Bisimulation. *J. ACM*, 42(2):458–487, 1995.
23. P. Piwowarski, M. Ohba, and J. Caruso. Coverage Measurement Experience during Function Test. In *Proc. of ICSE'93*, pages 287–301. IEEE, 1993.
24. G. Salaün, X. Etchevers, N. Palma, F. Boyer, and T. Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Proc. of SAC'12*, pages 1278–1283. ACM, 2012.