Automatic Test Set Generator with Numeric Constraints Abstraction for Embedded Reactive Systems: AUTSEG V2

Mariem Abdelmoula, Daniel Gaffé, and Michel Auguin

LEAT, University of Nice-Sophia Antipolis, CNRS
Email: Mariem.Abdelmoula@unice.fr
Email: Daniel.Gaffe@unice.fr
Email: Michel.Auguin@unice.fr

***Abstract*—AUTSEG is an automatic test set generator for embedded reactive systems. It automatically generates exhaustive test sets and allows to check safety properties of the tested system. A first version of AUTSEG has been initially designed for programs dealing with Boolean inputs and outputs. We present in this paper an extension of this tool called AUTSEG V2 to handle symbolic numeric data processing that provides more expressive and concrete tests of the system. To this end, we have developed a new library called superior linear decision diagrams (SupLDD) built on top of linear decision diagrams (LDD) library. This allows symbolic computation of system data while improving system verification (Determinism, Death sequences) and identifying all possible test cases. Our tool characterizes the system preconditions by numeric constraints to derive automatically the symbolic test cases using a backtracking operation. We demonstrate the application of AUTSEG V2 on an industrial example.**

Keywords–Test Sets; Synchronous Model; Pre-conditions; Numeric Data Processing; Backtrack; AUTSEG V2; SupLDD.

I. Introduction

Systems verification receives a particular interest today, especially for embedded reactive systems which have complex behaviors over time and which require long test sequences. This kind of systems is increasingly dominating safety critical domains such as nuclear industry, health insurance, banking, chemical industry, mining, avionics and online payment where failure could be disastrous. A practical solution in industry is to proceed using intensive test patterns in order to discover bugs, and increase confidence in the system, while researchers concentrate their efforts rather on formal verification. However, testing is obviously non exhaustive and formal verification is impracticable on real systems because of the combinatorial explosion nature of the states space.

AUTSEG [1] combines these two approaches to provide an automatic test set generator where formal verification ensures the automation in all phases of design, execution and test evaluation and help on get confidence in the consistency and relevance of tests. In a first version of AUTSEG, only Boolean inputs and outputs were supported while most of actual systems handle numeric data. Numeric data manipulation represents a big challenge for most of existing test generation tools due to the difficulty to express formal properties on those data using a concise representation. In our approach, we consider symbolic test sets which are thereby more expressive, safe and less complex than the concrete ones.

Therefore, we develop in this paper a new version of AUTSEG to take into account numeric data manipulation in addition to Boolean data manipulation. This was achieved by developing a new library for data manipulation called SupLDD. Prior automatic test sets generation methods have been consequently extended and adapted to this new numeric context. Symbolic data manipulations in AUTSEG V2 allow not only symbolic data calculations but also system verification (Determinism, Death sequences), and identification of all possible test cases without requiring the coverage of all the system states and transitions. Therefore, our approach bypasses in numerous cases the states space explosion problem. We besides defined a backtrack operation to exhibit significant test sets of the target system.

In the remainder of this paper, we briefly recall the principles of AUTSEG V1 and introduce the new version AUTSEG V2. The principles of data manipulation and its capabilities on tests generation and verification are presented in Section II. A case study is presented in Section III. We show in Section IV experimental results. Finally, we conclude the paper in Section V with some directions for future works.

II. AUTSEG V2 Description

A. Architectural Test Overview

We introduce in this section the principles of our automatic testing approach including data manipulation. Figure 1 shows five main operations including: i) the design of a global model of the system under test, ii) a quasi-flattening operation, iii) a compilation process, iv) a generation process of symbolic sequences mainly related to the symbolic data manipulation entity, v) and finally the backtrack operation to generate all possible test cases.

In this paper, we particularly focus on verification of embedded software controlling reactive systems behavior. The conception of such systems is generally based on the synchronous approach [2] that presents clear semantics to exceptions, delays and actions suspension. This notably reduces the programming complexity and favors the application of verification methods. In this context, we present the global model by hierarchical and parallel concurrent Finite States Machines (FSMs) based on the synchronous approach. The hierarchical machine describes the global system behavior, while parallel automata act as observers for control data of

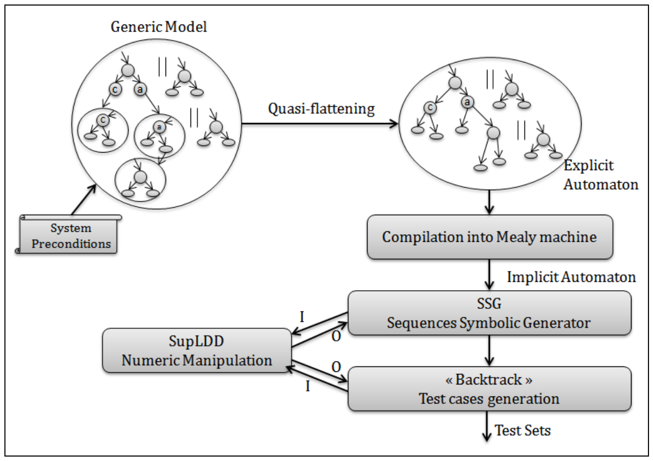


Figure 1. Global test process.

the hierarchical automaton. Our approach allows to test many types of a system at once. In fact, we present a single generic model for all types of the system, the specification of tests can be done later using particular Boolean variables called system preconditions (type of system, system mode, etc.). Hence, a specific test generation could be done at the end of test process through analysis of the system preconditions. This prevents to generate as many models as system types, which can highly limit the legibility and increase the risk of specification bugs.

A straightforward way to analyze a hierarchical machine is to flatten it first (by recursively substituting in a hierarchical FSM each super state with its associated FSM) and then apply on the resulting FSM a verification tool such as a model cheking tool. However, to analyze the global model, a full flattening of the hierarchical FSM is not required. Only the sequential hierarchical automata is flattened, the global structure remains parallel. In fact, flattening parallel FSMs explodes usually in number of states. Thus there is no need to flatten them, as we can compile them separately thanks to the synchronous approach [2], then concatenate them with the flat model retrieved at the end of the compilation process. This quasi-flattening operation allows to flatten the hierarchical automata and maintain the parallelism. This offers a simpler model, a faster compilation, and brings more flexibility to identify all possible evolutions of the system as detailed in the following steps.

Resulting flat automata and concurrent automata are then compiled separately into explicit Mealy machines, implicitly represented by a set of Boolean equations. Compilation results of these automata are concatenated at the end of this process. They are represented by a union of sorted equations rather than a Cartesian product of graphs to support the synchronous parallel operation and instantaneous diffusion of signals as required by the synchronous approach. Accordingly, a substantial reduction is brought on the size of the system model. Our compilation requires only $log_2(nbstates)$ registers, while classical works uses one register per state [3]. It allows also checking the determinism of all automata which ensures the persistence of the system behavior.

To supply numeric data manipulation in our tests, we developed SupLDD library offering symbolic means to characterize several preconditions by numeric constraints. It is sorely based on the potency of LDD library [4]. The symbolic representation of these preconditions shows an important role in the following operations of sequences symbolic generation and test cases generation "Backtrack". It evenly enhances system security by analyzing the constraints computations.

During the sequences symbolic generation operation, we automatically extract necessary preconditions which lead to specific, significant states of the system from generated sequences. Having defined the optimal preconditions for restricting the states space, we work locally on significant subspaces. This sequences generation process relies on the effective representation of the global model and the robustness of numeric data processing to generate the exhaustive list of possible sequences, avoiding therefore the manual and explicit presentation of all possible combinations of system commands.

Finally, the verification of the whole system behavior is performed by the manipulation of extracted preconditions from each significant subspace. Namely, we verify the execution context of each significant subspace. This verification is performed by the backtrack operation. It generates all possible test cases of the system under test. Specifically, it identifies all paths satisfying each final critical state preconditions to reach the root state.

We have already detailed in [1] the principles of the global model conception, the quasi-flattening operation and the compilation process. We will rather focus in the rest of this paper on the presentation of symbolic data manipulations and their capabilities to carry the symbolic sequences generation and the backtrack operation.

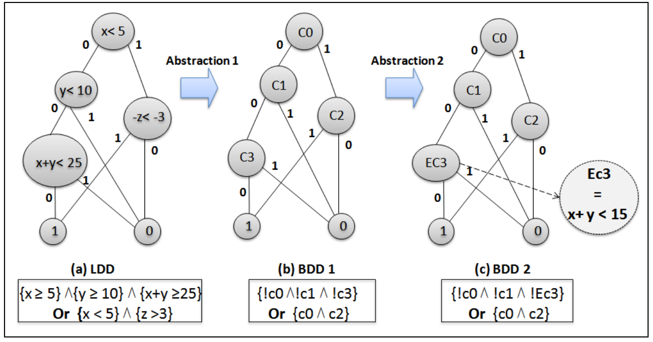
B. Symbolic data manipulation

1) Related work: Since 1986, Binary Decision Diagrams (BDDs) have successfully emerged to represent Boolean functions for formal verification of systems with large states space. BDDs, however, cannot represent quantitative information such as integers and real numbers. Variations of BDDs have been proposed thereafter to support symbolic data manipulations that are required for verification and performance analysis of systems with numeric variables. For example, Multi-Terminal Binary Decision Diagrams (MTBDDs) [5] are a generalization of BDDs in which there can be multiple terminal nodes, each labeled by an arbitrary value. However, the size of nodes in an MTBDD can be exponential (2^n) for systems with large range of values. To support a larger number of values, Yung-Te Lai has developed Edge-Valued Binary Decision Diagrams (EVBDDs) [6] as an alternative to MTBDDs to offer a more compact form. EVBDDs associate multiplicative weights with the true edges of an EVBDD function graph to allow an optimal sharing of subgraphs. This suggests a linear evolution of non-terminal nodes size rather than an exponential one for MTBDDs. However, EVBDDs are limited to relatively simple calculations units, such as adders and comparators, implying a high cost per node for complex calculations such as $(X \times Y)$ or (2^X).

To overcome this exponential growth, Binary Moment Diagrams (BMDs) [7], another variation of BDDs, have been specifically developed for arithmetic functions considered as linear functions with Boolean inputs and integer outputs to perform a compact representation for integer encodings and operations. They integrate a moment decomposition principle giving way to two sub-functions representing the two moments (constant and linear) of the function, instead of a decision. This representation was later extended to Multiplicative Binary Moment Diagrams (*BMDs) [8] to include weights on edges allowing to share common sub-expressions. These edges weights are multiplicatively combined in a *BMD, in contrast to the principle of addition in an EVBDD. Thus, the following arithmetic functions $X + Y$, $X - Y$, $X \times Y$, 2^X show representations of linear size. Despite their significant success in several cases, handling edges weights in BMDs and *BMDs is a costly task. Moreover, BMDs are unable to verify the satisfiability property, and functions outputs are non divisible integers to separate bits, causing a problem for applications with output bit analysis. BMDs and MTBDDs were combined by Clarke and Zhao in Hybrid Decision Diagrams (HDDs) [9]. But, all of these diagrams are restricted to materials arithmetic circuits check and not suitable for the verification of software systems specifications.

Within the same context of arithmetic circuits check, Taylor Expansion Diagrams (TEDs) [10] have been introduced to supply a new formalism for multi-values polynomial functions providing a more abstract, standard and compact design representation, with integer or discrete inputs and outputs values. For an optimal fixed order of variables, the resulting graph is canonical and reduced. Unlike the above data structures, TED is defined on a non-binary tree. In other words, the number of child nodes depends on the degree of the relevant variable. This makes TED a complex data structure for particular functions such as (a^x). In addition, the representation of the function $(x < y)$ is an important issue in TED. This is particularly challenging for the verification of most software systems specifications. In this context, Decision Diagrams for Difference logic (DDDs) [11] have been proposed to present functions of first order logic by inequalities of the form $\{x - y \leq c\}$ or $\{x - y < c\}$ with integer or real variables. The key idea is to present these logical formulas as BDD nodes labeled with atomic predicates. For a fixed variables order, a DDD representing a formula f is no larger than a BDD of a propositional abstraction of f. It supports as well dynamic programming by integrating an algorithm called QELIM based on Fourier-Motzkin elimination [12]. Despite their proved efficiency in verifying timed systems [13], the difference logic in DDDs is too restrictive in many program analysis tasks. Even more, dynamic variable ordering (DVO) is not supported in DDDs. To address those limitations, LDDs [4] extend DDDs to full Linear Arithmetic by supporting an efficient scheduling algorithm and a QELIM quantification. They are BDDs with non terminal nodes labeled by linear atomic predicates satisfying a scheduling theory and local constraints reduction. Data structures in LDDs are optimally ordered and reduced by considering the several implications of all atomic predicates. LDDs have the possibility of computing arguments that are not fully reduced or canonical for most LDD operations. This suggests the use of various reduction heuristics that trade off reduction potency for calculations cost.

2) SupLDD: We summarize from the above data structures that LDD is the most relevant work for data manipulation in our context. We present in this section a new library for data manipulation called SupLDD founded on LDD basis. Figure 2 shows an example of representation in SupLDD of the arithmetic formula $F1 = \{(x \geq 5) \wedge (y \geq 10) \wedge (x + y \geq 25)\} \vee \{(x < 5) \wedge (z > 3)\}$. Nodes of this structure are labeled by the linear predicates $\{(x < 5); (y < 10); (x + y < 25); (-z < -3)\}$ of formula F1, where the right branch evaluates its predicates to 1 and the left branch evaluates its predicates to 0. In fact, the choice of a particular comparison operator within the 4 possible operators $\{<, \leq, >, \geq\}$ is not important since the 3 other operators can always be expressed from the chosen operator: $\{x < y\} \Leftrightarrow \{NEG(x \geq y)\}; \{x < y\} \Leftrightarrow \{-x > -y\}$ and $\{x < y\} \Leftrightarrow \{NEG(-x \leq -y)\}$.


x< 5
y< 10
-z< -3
x+y< 25
0
1
Abstraction 1
Abstraction 2
C0
C1
C2
C3
EC3
Ec3 = x+ y < 15
(a) LDD
{x ≥ 5} ∧{y ≥ 10} ∧ {x+y ≥25} Or {x < 5} ∧ {z >3}
(b) BDD 1
{!c0 ∧!c1 ∧ !c3} Or {c0 ∧ c2}
(c) BDD 2
{!c0 ∧ !c1 ∧ !Ec3} Or {c0 ∧ c2}
Figure 2. Representation in SupLDD of F1.

We show in Figure 2.b that the representation of F1 in SupLDD has the same structure as a representation in BDD that labels its nodes by the corresponding Boolean variables $\{C0; C1; C2; C3\}$ to each SupLDD predicate. But, a representation in SupLDD is more advantageous. In particular, it ensures the numeric data evaluation and manipulation of all predicates along the decision diagram. This furnishes a more accurate and expressive representation in Figure 2.c than the original BDD representation. Namely, the Boolean variable C3 is replaced by EC3 which evaluates the corresponding node to $\{x+y < 15\}$ instead of $\{x+y < 25\}$ taking into account prior predicates $\{x < 5\}$ and $\{y < 10\}$. Besides, SupLDD relies on an efficient T-atomic scheduling algorithm [4] that makes compact and non-redundant diagrams for SupLDD where a node labeled for example by $\{x \leq 15\}$ never appears as a right child of a node labeled by $\{x \leq 10\}$. As well, nodes are ordered by set of atoms $\{x, y, etc.\}$ where a node labeled by $\{y < 2\}$ never appears between two nodes labeled by $\{x < 0\}$ and $\{x < 13\}$. Further, SupLDD diagrams are optimally reduced including the LDD reduction rules. First, the QELIM quantification introduced in LDDs allows the elimination of multiples variables: For example, the QELIM quantification of the expression $\{(x-y \leq 3) \wedge (x-t \geq 8) \wedge (y-z \leq 6) \wedge (t-k \geq 2)\}$ eliminates the intermediate variables *y* and *t* and generates the simplified expression $\{(x - z \leq 9) \wedge (x - k \geq 10)\}$. Second, the LDD high implication [4] rule allows to get the smallest geometric space: For example the simplification of the expression $\{(x \leq 3) \wedge (x \leq 8)\}$ in high implication turns to the single term $\{x \leq 3\}$. Finally, the LDD low implication [4] rule generates the largest geometric space where the expression $\{(x \leq 3) \wedge (x \leq 8)\}$ becomes $\{x \leq 8\}$.

SupLDD operations- SupLDD operations are primarily generated from basic LDD operations [4]. They are simpler and more adapted to our needs. We present functions to manipulate inequalities of the form $\{\sum a_i x_i \leq c\}$; $\{\sum a_i x_i < c\}$; $\{\sum a_i x_i \geq c\}$; $\{\sum a_i x_i > c\}$; where $\{a_i, x_i, c \in Z\}$. Given two inequalities I_1 and I_2, the main operations in SupLDD include:

- SupLDD conjunction (*I1*, *I2*): This absolutely corresponds to the intersection on *Z* of sub-spaces representing *I1* and *I2*.
- SupLDD disjunction (*I1*, *I2*): As well, this operation absolutely corresponds to the union on *Z* of sub-spaces representing *I1* and *I2*.

Accordingly, all the space *Z* can be represented by a union of two inequalities $\{x \leq a\} \cup \{x > a\}$. As well, the empty set can be inferred from the intersection of inequalities $\{x \leq a\} \cap \{x > a\}$.

- Equality operator $\{\sum a_i x_i = c\}$: It is defined by the intersection of two inequalities $\{\sum a_i x_i \leq c\}$ and $\{\sum a_i x_i \geq c\}$.
- Resolution operator: It simplifies arithmetic expressions using QELIM quantification, and both low and high implication rules introduced in LDD. For example, the QELIM resolution of $\{(x-y \leq 3) \wedge (x-t \geq 8) \wedge (y-z \leq 6) \wedge (x-t \geq 2)\}$ gives the simplified expression $\{(x-z \leq 9) \wedge (x-t \geq 8) \wedge (x-t \geq 2)\}$. This expression can be more simplified to $\{(x-z \leq 9) \wedge (x-t \geq 8)\}$ in case of high implication and to $\{(x-z \leq 9) \wedge (x-t \geq 2)\}$ in case of low implication.
- Reduction operator: It solves an expression *A* with respect to an expression *B*. In other words, if *A* implies *B*, then the reduction of *A* with respect to *B* is the projection of *A* when *B* is true. For example, the projection of *A* $\{(x-y \leq 5) \wedge (z \geq 2) \wedge (z-t \leq 2)\}$ with respect to *B* $\{x-y \leq 7\}$ gives the reduced set $\{(z \geq 2) \wedge (z-t \leq 2)\}$.

We report in this paper on the performance of these functions to enhance our tests. More specifically, by means of SupLDD library, we present next an extension of Sequences Symbolic Generation operation initially presented in AUTSEG V1 to integrate data manipulation and generate more significant and expressive sequences. Moreover, we track and analyze tests execution to spot the situations where the program violates its properties. In the other hand, our library ensures the analyze of generated sequences context to carry the backtrack operation and generate all possible test cases.

C. *Sequences Symbolic Generation (SSG)*

In this version, we take into account data calculations within the sequences generation process. Let's recall the principles of SSG in AUTSEG V1. In fact, our approach is primarily designed to test systems running iterative commands. In this context, we confine only on significant sub-spaces representing each command of the system instead of considering all the states space. Indeed, we test all the system commands, but one command is tested at once. This restriction was done by characterizing all preconditions defining the execution context in each subspace. Hence, the major complex calculation is intended to be locally done in each significant subspace avoiding the states space combinatorial explosion problem. Figure 3 shows this efficient representation of the system behavior. It presents a repetition of a subspace pattern representing a specific system command instead of an infinite tree if we typically imagine all possible combinations of the system iterative commands.

Each state in the subspace is specified by 3 main variables: symbolic values of the program variables, path condition and command parameters (next byte-code to be executed). The path condition represents preconditions that should be satisfied by the symbolic values to successfully progress the execution of the current path. We particularly define two types of preconditions:

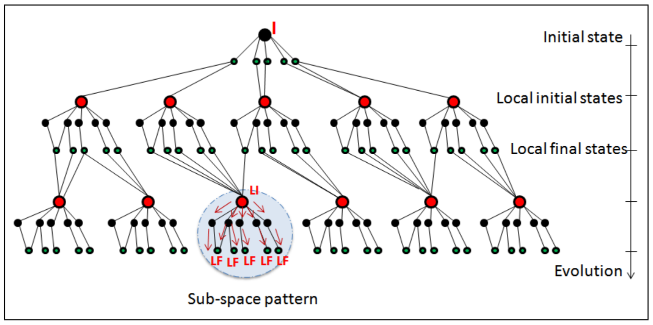


Figure 3. AUTSEG V2 Model Representation.

- Boolean global preconditions that define the execution context of a given command. They states the list of commands that should be executed before. They arise as command output if this latter is properly executed.
- Numeric local preconditions that define numeric constraints on commands parameters. They are presented and manipulated by SupLDD functions.

We have explained in details in the first version of AUTSEG the SSG operation. We have applied BDD-analysis to generate all possible paths from a Local Initial state (*LI*) to reach Local Final states (*LF*) of the tested subspace. As well, necessary preconditions are extracted from this subspace check. We extend in this paper the SSG operation to integrate data manipulations. We apply SupLDD analysis on numeric local preconditions to check if the tested system is safe. We firstly check if there are erroneous sequences. To this end, we apply the SupLDD conjunction function on all extracted numeric preconditions within the analyzed path. If the result of this conjunction is null, the analyzed sequence is then impossible and should be rectified! Second, we check the determinism of the system behavior. To this end, we verify if the SupLDD conjunction of all outgoing transitions from each state is empty. In other words, we verify if the SupLDD disjunction of all outgoing transitions from each state is equal to all the space covering all possible system behaviors.

Contrary to the classical sequences generator, our tool constantly generates a tree of pure future states, thus preventing loops from occurring. Namely, previous states always converge to the global initial state. This approach easily favors the backtrack execution.

D. *Backtrack operation*

Once the necessary preconditions are extracted, a next step is to backtrack paths from each final critical state until the initial state finding the sequence fulfilling these preconditions. This operation is carried by robust calculations on SupLDD and the compilation process which kept enough knowledge to find later the previous states. It includes two main actions: a global backtrack and a local backtrack. Let's consider the SSG representation in Figure 3, if we take into account *LF* as a critical final state *FS* of the tested system, the global backtrack operation is to find the list of commands that should be executed before the tested command. Figure 4 details this operation: Given the global extracted preconditions (*GP1*,*GP2*, etc.) from the SSG operation at this level (Final state *FS* of command *C1*), we search in the global actions table for actions (Commands *C2* and *C3*) that emit each parsed global precondition. Next, we put on a list *SL* the states that trigger each identified action (*SL*= $\{C2, C3\}$). This operation is iteratively executed on all found states (*C2,C3*) until reaching the root state *I* with zero preconditions (*C4* with zero preconditions).

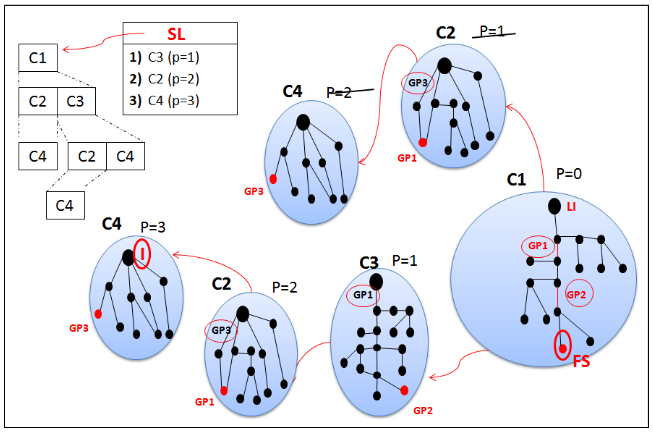


Figure 4. Global Backtrack.

As many commands can share the same global preconditions (*C1* and *C3* share the same precondition *GP1*), the identified states can be repeated on *SL* (*C2* and *C4* are repeated on *SL*). To manage this redundancy, we allocate a priority *P* to each found state where each state of priority *P* should precede the state of priority *P+1*. More specifically, if an identified state already exists in *SL*, then its priority is incremented by 1 (Priority of *C2* and *C4* are incremented by 1). By the end of this operation, we obtain the list *SL* (*SL*= { *C3,C2,C4* }) of final states refering to subspaces that should be traced to reach *I*.

A next step is to execute a local backtrack on each identified subspace (*C1,C3, C2,C4*) starting from the state with the lowest priority and so on to trace the final path from *FS* to *I*. The sequence from *I* to *FS* is an example of a good test set. Figure 5 presents an example of local backtrack in command *C3*. In fact, during the SSG operation each state *S* was labeled by (1) a Local numeric Precondition (*LP*) presenting numeric constraints that should be satisfied on its ongoing transition and (2) a Total Local numeric precondition (*TL*) that presents the conjunction of all *LP* along the executed path from *I* to *S*. To execute the local backtrack, we start from the ongoing transition *PT* to *FS* to find a path that satisfy the backtrack precondition *BP* initially defined by *TL*. If the backtrack precondition is satisfied by the total precondition $\{TL \geq BP\}$, then if the local precondition *LP* of the tested transition is not null, So we remove this verified precondition *LP* from *BP* by applying the SupLDD projection function. Next, we move to the amount state of *PT* and test its ongoing transitions, etc. However, if $\{TL < BP\}$, we move to the test of other ongoing transitions to find the transition from which *BP* can be satisfied. This operation is iteratively executed until reaching the initial state on which the backtrack precondition

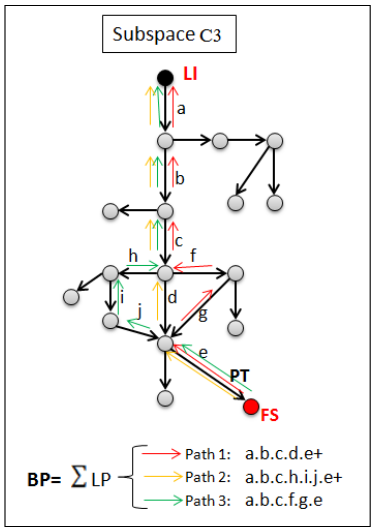


Figure 5. Local Backtrack.

is null (fully satisfied). In short, if the context is verified, the generated sequence is considered correct. At the end of this process, we join all identified paths from each traced subspace according to the given priority order from the global backtrack operation.

III. USE CASE

To illustrate our approach, we studied the case of a contactless smart card for the transportation sector manufactured by ASK company [14], a world leader in contactless smart card technology. We specifically target the verification of the card's functionality and security features. Overall, security of such systems is critical: it can concern cards for access security, banking, ID, etc. Cards complexity makes it difficult for a human to identify all possible sensitive situations or to validate it by classical methods. We need approximately 500 000 years to test the first 8 bytes if we consider a classical Intel processor able to generate 1000 test sets per second. As well, combinatorial explosion of possible modes of operation makes it nearly impossible to attempt a comprehensive simulation. The problem is exacerbated when the system integrates numeric data processing. We have already studied this use case within the first version of AUTSEG, but processing numeric variables was ignored. We rather show in this section real tests with AUTSEG V2 taking into account the complexity of data manipulation.

The smart card operation is defined by a transport standard called Calypso that presents 33 commands. The succession of these commands (e.g., Open Session, SV Debit, Get Data, Change Pin) gives the possible scenarios of card operation. We designed the generic model of the studied card by 52 interconnected automata including 765 states. Forty three of them form a hierarchical structure. The remaining automata operate in parallel and act as observers to control the global context of hierarchical automaton (Closed Session, Verified PIN, etc.). We choose to use Light Esterel (light version of SyncChart) [15], a synchronous graphical model that integrates high-level concepts of synchronous languages in an expressive graphical formalism. We show in Figure 6 a small part of our model representing the command Open Session. Each command in Calypso is presented by an APDU (Application Protocol Data Unit) that presents the next byte-code to be executed (CLA,INS,P1,P2, etc.). We expressed these parameters by SupLDD local preconditions on various transitions. For instance, AUTSEGINT$(h10 < P1 < h1E)$ means that the corresponding transition can only be executed if $(10 < P1 < 30)$. Back-Autseg-Open-Session and Back-Autseg-Verify-PIN are examples of global preconditions that appear as outputs of respectively Open Session and Verify PIN commands when they are correctly executed. They appear also as inputs for other commands as SV Debit command to denote that the card can be debited only if the PIN code is correct and a session is already open. Autseg-Contact-mode is an example of system precondition specifying that Open Session command should be executed in a Contactless Mode.

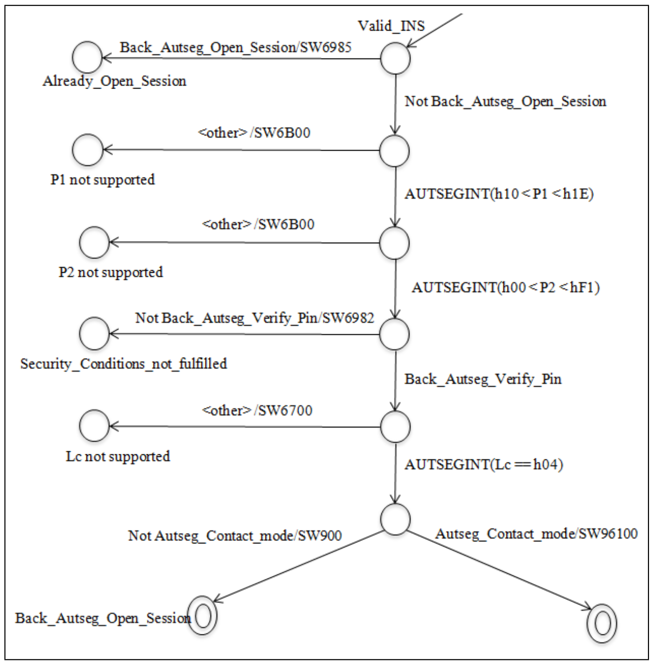


Figure 6. Open Session command.

IV. Experimental Results

In this section, we show experimental results of applying our tool to the contactless transportation card. We intend to test the security of all possible combinations of 33 commands of the Calypso standard. This validation process is extremely important to determine whether the card correctly meets its specifications. Each command in the Calypso standard is encoded on a minimum of 8 bytes. We conducted our experiments on a PC with Intel Dual Core Processor, 2 GHz and 8 GB RAM. We have already shown in a previous work [1] the successful application of our quasi-flattening process on the smart card hierarchical model. Compared to classical works, we have moved from 9.6 10^{24} states in the designed model to only 256 per branch of parallel. Then, due to the compilation process, we have moved from 477 registers to only 22.

We present in this paper more interesting results on sequences generation and test coverage with data processing. The curve denoted C1 in Figure 7 shows an exponential evolution of the number of generated sequences versus the number of tested bytes. This corresponds to a classical testing method that browses all possible paths of the card model without any restriction. We are not even able to test more than 2 commands of the model. Our model explodes by 13 bytes generating 3,993,854,132 possible sequences. A second test applies AUTSEG V1 on the card model represented in the same manner as Figure 3. Results shows in curve C2 a lower evolution that stabilizes at 10 steps and 1784 paths, allowing for coverage of all states of the tested model. More interesting results are shown in curve C3 by AUTSEG V2 tests. Our approach enables coverage of the global model in a substantially short time (few seconds). It allows separately testing 33 commands (all the system commands) in only 21 steps, generating a total of solely 474 paths. Covering all states in only 21 steps, our results demonstrate that we test separately one command (8 bytes) at once in our approach thanks to the backtrack operation. The additional steps (13 bytes) correspond to the test of system preconditions (e.g., Autseg-Contact-mode, etc.), global preconditions (Back-Autseg-Open-Session, etc.) and other local preconditions (e.g., AUTSEGINT$(h00 \leq buffer-size \leq hFF)$). Whereas, only fewer additional steps (2 bytes) are required within the first version of AUTSEG that stabilizes at 10 steps. This difference proves a complete evaluation of system constraints by our new version of AUTSEG performing therefore more expressive and reel tests: we integrate a better knowledge of the system.

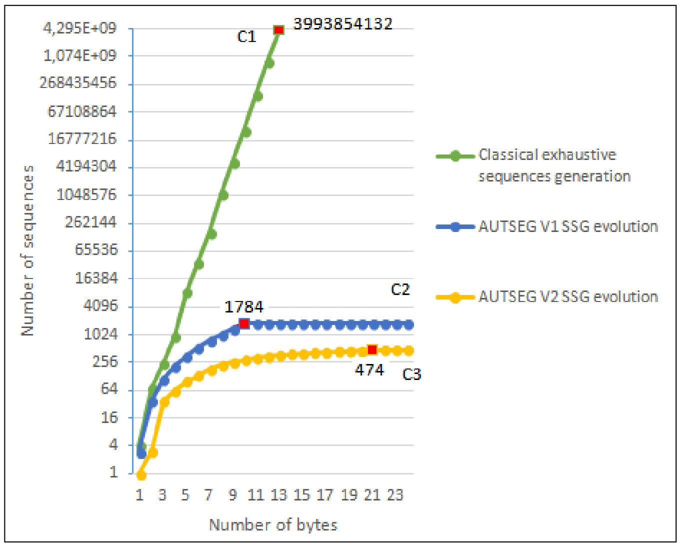


Figure 7. SSG evolutions.

Curve C4 in Figure 8 exhibits results of AUTSEG V2 tests simulated with 3 anomalies on the smart card model. We note less generated sequences by the 5 steps. We obtain a total of 460 sequences instead of 474 at the end of tests. 14 sequences are removed since they are unfeasible (dead sequences) by SupLDD calculations. Indeed, the SupLDD conjunction of parsed local preconditions AUTSEGINT$(01h \leq RecordNumber \leq 31h)$ and AUTSEGINT$(RecordNumber \geq FFh)$ within a same path is null presenting an over-specification example (anomaly) of the Calypso standard that should be revised.

We show in Figure 9 an excerpt of generated sequences

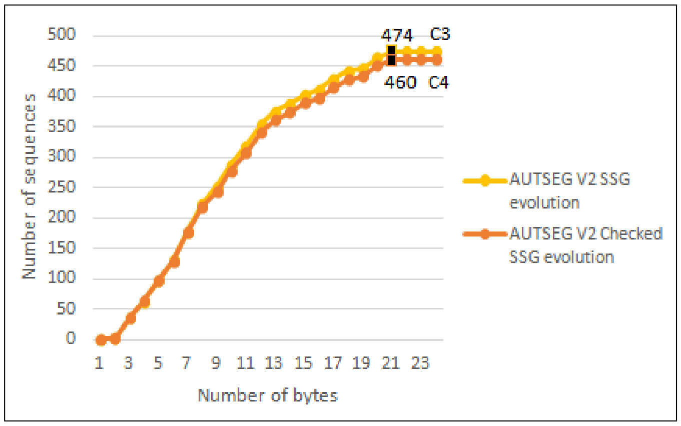


Figure 8. AUTSEG V2 SSG evolutions.

by AUTSEG V2 detecting another type of anomaly: an under-specification in the card behavior. The *Incomplete Behavior* message reports a missing action on a tested state of Update-Binary command. Indeed, two actions are defined ($Tag = 54h$) and ($Tag = 03h$) at this state. All states where Tag is different from *84* and *3* are missing. We can automatically spot such problems by checking for each parsed state if the union SupLDD-Or of all outgoing transitions is equal to all the space. Once, this property is always true, then the smart card behavior is proved deterministic.

```
Update Binary command Test
-------------------------------------------
Sequence:

-AUTSEGINT(CLA==00h OR CLA==94h) -->
-AUTSEGINT(INS ==D7h) -->
-Back-Autseg-Open-Session AND not Wrong Key
-AUTSEGINT(P1==00h) -->
-AUTSEGINT(P2>=00h AND P2<=1Eh) -->
-AUTSEGINT(SFI>=00h AND SFI<=FFh)
-AUTSEGINT <other> -->
-AUTSEGINT(EFTYPE ==01h) -->
-AUTSEGINT(Lc>=07h AND Lc<=FFh) -->
-AUTSEGINT(Tag==54h) -->
----> Incomplete Behavior!

Post transitions:
Parse expression AUTSEGINT(Tag ==54h)
Parse expression AUTSEGINT(Tag ==03h)
-------------------------------------------
```

Figure 9. Smart Card Under-specification.

As explained before, we get the execution context of each generated sequence at the end of this operation. The next step is then to backtrack all critical states of the Calypso standard (all final states of 33 commands). We show in Figure 10 a detailed example of backtrack from the final state of SV Undebit command that emit SW6200 code.

We identify from the global extracted preconditions Back-Autseg-Open-Session and Back-Autseg-Get-SV the list of commands (Open Secure Session and SV Get) to be executed before. Then, we look recursively for all global preconditions

```
Backtrack from state SV Undebit
Postponed Response Data SW6200
-------------------------------------------
Backtrack Sequence:

-Back-Autseg-Open-Session --> SW6200
-not Session memory full -->
-AUTSEGINT(Amount<0) -->
-tick --> Autseg Memorize(Amount)
-AUTSEGINT(Lc==14h) -->
-Present SV -->
-Back-Autseg-Get-SV -->
-AUTSEGINT(INS==BCh) -->
-AUTSEGINT(CLA==FAh) -->

Backtrack var is Back-Autseg-Get-SV
Backtrack var is Back-Autseg-Open-Session
Backtrack var is Back-Autseg-Verify-PIN
Prior commands to execute 3
-------------------------------------------
```

Figure 10. SV Undebit Backtrack.

of each identified command to trace the complete path to the initial state of Start command. We observe from the results that Verify PIN command should proceed the Open Secure Session command. So, the final backtrack path is to trace (local backtrack) the identified commands respectively SV Undebit, SV Get, Open Secure Session et Verify PIN using local preconditions of each command.

At the end of this process, we generate automatically 5456 test sets that cover the entire behavior of the studied smart card. While, industrials take much more time to solely generate manually 520 test sets covering 9,5% of our tests as shown in Figure 11.

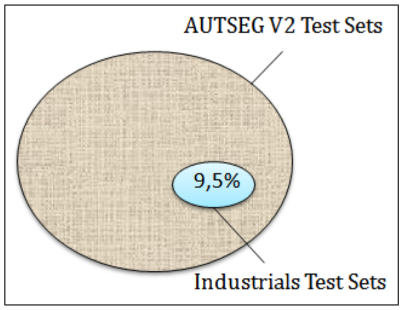


Figure 11. Tests Coverage.

V. Conclusion

We have proposed an extension of AUTSEG to integrate data manipulations. For this purpose, we have developed a new library called SupLDD that supplies numeric data manipulations and takes advantages of the symbolic encoding scheme of BDDs. This library allows not only symbolic calculations of system data but also the verification of the system behavior. Our method is practical and performs well, even with large models where the risk of combinatorial explosion of states space is important. Our experiments confirm that our tool provides more expressive and significant tests covering all possible system evolutions in a short time. More generally, our tool including the SupLDD calculations can be applied to many numeric systems as they could be modeled by FSMs handling integer variables.

Since SupLDD is implemented on top of a simple BDD package. We aim in a future work to rebuild SupLDD on top of an efficient implementation of BDDs with complement edges [16] to get a better optimization of our library. Another interesting contribution would be to integrate SupLDD in data abstraction of CLEM [15]. More details about these future works are presented in [17].

References

[1] M. Abdelmoula, D. Gaffé, and M. Auguin, "Autseg: Automatic test set generator for embedded reactive systems," in Testing Software and Systems, 26th IFIP International Conference,ICTSS, ser. Lecture Notes in Computer Science. Madrid, Spain: springer, September 2014, pp. 97–112.

[2] C. André, "A synchronous approach to reactive system design," in 12th EAEEIE Annual Conf., Nancy (F), May 2001, pp. 349–353.

[3] I. Chiuchisan, A. D. Potorac, and A. Garaur, "Finite state machine design and vhdl coding techniques," in 10th International Conference on development and application systems. Suceava, Romania: Faculty of Electrical Engineering and Computer Science, 2010, pp. 273–278.

[4] S. Chaki, A. Gurfinkel, and O. Strichman, "Decision diagrams for linear arithmetic." in FMCAD. IEEE, 2009, pp. 53–60.

[5] M. Fujita, P. C. McGeer, and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation," Form. Methods Syst. Des., vol. 10, no. 2-3, Apr. 1997, pp. 149–169.

[6] Y.-T. Lai and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification," in Proceedings of the 29th ACM/IEEE Design Automation Conference, ser. DAC'92. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 608–613.

[7] R. E. Bryant and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference, ser. DAC '95. New York, NY, USA: ACM, 1995, pp. 535–541.

[8] L. Arditi, "A bit-vector algebra for binary moment diagrams," I3S, Sophia-Antipolis, France, Tech. Rep. RR 95–68, 1995.

[9] E. Clarke and X. Zhao, "Word level symbolic model checking: A new approach for verifying arithmetic circuits," Pittsburgh, PA, USA, Tech. Rep., 1995.

[10] M. Ciesielski, P. Kalla, and S. Askar, "Taylor expansion diagrams: A canonical representation for verification of data flow designs," IEEE Transactions on Computers, vol. 55, no. 9, 2006, pp. 1188–1201.

[11] J. Møller and J. Lichtenberg, "Difference decision diagrams," Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, Aug. 1998.

[12] A. J. C. Bik and H. A. G. Wijshoff, Implementation of Fourier-Motzkin Elimination. Rijksuniversiteit Leiden. Valgroep Informatica, 1994.

[13] P. Bouyer, S. Haddad, and P.-A. Reynier, "Timed petri nets and timed automata: On the discriminating power of zeno sequences," Inf. Comput., vol. 206, no. 1, Jan. 2008, pp. 73–107.

[14] "Ask," [Retrieved: 16-October-2015]. [Online]. Available: http://www.ask-rfid.com/

[15] A. Ressouche, D. Gaffé, and V. Roy, "Modular compilation of a synchronous language," in Soft. Eng. Research, Management and Applications, best 17 paper selection of the SERA'08 conference, R. Lee, Ed., vol. 150. Prague: Springer-Verlag, August 2008, pp. 157–171.

[16] K. Brace, R. Rudell, and R. Bryant, "Efficient implementation of a bdd package," in Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE, June 1990, pp. 40–45.

[17] M. Abdelmoula, "Automatic test set generator with numeric constraints abstraction for embedded reactive systems," Ph.D. dissertation, Published in "Génération automatique de jeux de tests avec analyse symbolique des données pour les systèmes embarqués", Sophia Antipolis University, France, 2014.