Combining SLiVER with CADP to Analyze Multi-agent Systems

Luca Di Stefano[1,2](✉), Frédéric Lang[3], and Wendelin Serwe[3]

[1] Gran Sasso Science Institute (GSSI), L'Aquila, Italy
luca.distefano@gssi.it
[2] IMT School of Advanced Studies, Lucca, Italy
[3] Univ. Grenoble Alpes, Inria, CNRS,
Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), LIG,
38000 Grenoble, France

Abstract. We present an automated workflow for the analysis of multi-agent systems described in a simple specification language. The procedure is based on a structural encoding of the input system and the property of interest into an LNT program, and relies on the CADP software toolbox to either verify the given property or simulate the encoded system. Counterexamples to properties under verification, as well as simulation traces, are translated into a syntax similar to that of the input language: therefore, no knowledge of CADP is required. The workflow is implemented as a module of the verification tool SLiVER. We present the input specification language, describe the analysis workflow, and show how to invoke SLiVER to verify or simulate two example systems. Then, we provide details on the LNT encoding and the verification procedure.

1 Introduction

Multi-agent systems are composed of standalone computational units, the agents, that interact with each other and with an external environment. Computation within each agent may be a composition of multiple interleaving processes. The agents may also interleave their executions and interact with each other, possibly through asynchronous interaction patterns. As a consequence, multi-agent systems typically feature extremely large state spaces, which makes them hard to design and reason about.

Therefore, there is a need for languages that allow to specify these systems in a concise and intuitive fashion, as well as tools that can certify or increase confidence in the correctness of such specifications. This need is felt far beyond the multi-agent community, as agent-based models are gaining popularity in economics [13,29], social sciences [3,4], and many other research fields. However, the development of tools for such new languages may be a daunting task, as it

Work partially funded by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems).

© IFIP International Federation for Information Processing 2020
Published by Springer Nature Switzerland AG 2020
S. Bliudze and L. Bocchi (Eds.): COORDINATION 2020, LNCS 12134, pp. 370–385, 2020.
https://doi.org/10.1007/978-3-030-50029-0_23

must keep pace with both the evolution of the language and the state of the art in formal analysis of systems.

An alternative solution is to encode a system specification into an existing language, and reuse mature tools for that language to analyze the encoded system. An example of this approach is given by SLiVER, a prototype tool for the automated verification of multi-agent systems that are described in the simple specification language LAbS [10].[1] The tool is highly modular: it exploits the formal semantics of LAbS to encode the input system into an *emulation program* in a given language, through a structural translation procedure, and verifies the emulation program with off-the-shelf tools for that language to reach a verdict on the correctness of the input system. Previously [10], SLiVER only generated sequential C programs and verified them through bounded model checking [5], by using tools such as 2LS [8], CBMC [9], and ESBMC [14] as back ends.

In this paper, instead, we present a new analysis workflow based on process-algebraic tools. Namely, we choose the process calculus LNT [17] as the target language, and CADP [16] as the back end analysis tool.[2] The workflow is implemented as a SLiVER module and can verify both *invariance* properties (i.e., all reachable states satisfy a given formula) and *inevitable reachability* ones (i.e., all executions lead to a state where the given formula is satisfied), over the full state space of the input system. Furthermore, we can use the same workflow to *simulate* the evolution of the system and return a set of execution traces. This is the first SLiVER module that supports simulation. These two approaches may complement each other: even though simulation can rarely lead to strong conclusions about the correctness of a system [31], it is a valuable design aid and can provide quick feedback even on very large systems.

The rest of the paper is organized as follows. Section 2 briefly describes the specification language LAbS supported by SLiVER through an example, and contains an overview of LNT and CADP. Section 3 introduces the analysis workflow and its implementation as a SLiVER module, and provides usage examples. In Sect. 4 we describe in further detail how the tool generates emulation programs, and in Sect. 5 we explain how it performs property verification through model checking of such programs. Finally, we discuss related work in Sect. 6 and provide concluding remarks in Sect. 7.

2 Background

System Specifications. LAbS [10] is a domain-specific language to describe the *behavior* of agents in a multi-agent system. A behavior is made of basic *actions*, which tell the agent to assign a value to a variable. There are three kinds of assignments: to an internal variable, denoted by $x \leftarrow E$ (where x is a variable identifier and E a value expression); to a shared variable, denoted by $x \dashleftarrow E$; and to a *stigmergic* variable, denoted by $x \leftsquigarrow E$. Stigmergic variables are a distinguishing feature of LAbS. Their value is bound to a timestamp and stored

[1] A Linux release of SLiVER is available at https://git.io/sliver-tool.
[2] CADP is available at http://cadp.inria.fr.

on a decentralized data structure, allowing agents to share their knowledge with the rest of the system by exchanging asynchronous messages [26]. In brief, agents send *propagation* messages after updating a stigmergic variable. A propagation message contains the name of this variable, its new value, and its associated timestamp. An agent that receives a message checks whether its timestamp is newer than the local one for the same variable. If this is the case, the local value and timestamp are overwritten by the received ones; furthermore, the receiver will in turn propagate this new value to others. Otherwise, the message is simply discarded. Agents also send *confirmation* messages after reading the value of a stigmergic variable (i.e., by using it as part of a value expression). The contents of a confirmation message are the same as those of a propagation message. However, a receiver of a confirmation message that stores a value with a higher timestamp will react by propagating its own value. This mechanism facilitates the spread of up-to-date values through the system.

A single action may specify multiple assignments to variables of the same kind: for instance, an assignment to multiple internal variables is denoted by $x_1, \ldots, x_n \leftarrow E_1, \ldots, E_n$. Multiple assignments to variables of different kinds (e.g., an internal one and a shared one) are not allowed. Actions may be composed with traditional process-algebraic operators: sequential composition (;), nondeterministic choice (+), interleaving (|), and calls to other behaviors (possibly including recursive calls). Furthermore, a behavior B may be guarded by a condition g (denoted as $g \rightarrow B$), meaning that the agent may start behaving as B only if g holds.

SLiVER takes as input a system specification in a machine-readable version of LAbS, which is extended with constructs to specify the property of interest and the initial state of the system through (possibly nondeterministic) variable initialization expressions. Furthermore, the input format allows to parameterize systems in one or more external variables.

Figure 1a shows an example specification describing the well-known *dining philosophers* scenario. The system is parameterized in the number `_n` of agents (line 2), and features an array `forks`, which is shared by all the agents and whose elements are all initialized to 0 (line 3: the set of shared variables within a system is called its *environment*). Each element of the array models a fork: a value 0 means that the fork is available, while a value 1 means that it is currently held by one of the agents. The (recursive) behavior of the agents is specified at lines 10–21. Each agent repeatedly tries to acquire two forks, by checking and updating the elements `id` and `(id+1)%_n` of the array `forks`. The special variable `id` has a different value for each agent, and `%` denotes the modulo operator. After acquiring both forks, the agent releases them and starts over. Each agent maintains an internal variable `status`, initially set to 0, which describes its current situation (line 8: the set of internal variables of an agent is called its *interface*). When `status` is set to either 0, 1, or 2, it denotes the number of forks currently held by the agent. When `status` is set to 3, it means that the agent has just released one fork and is going to release the other one during its next action. Lastly,

```
 1  system {
 2    extern = _n
 3    environment = fork[_n]: 0
 4    spawn = Phil: _n
 5  }
 6
 7  agent Phil {
 8    interface = status: 0
 9
10    Behavior =
11      fork[id] = 0 ->
12        fork[id] <-- 1;
13        status <- 1;
14        fork[(id+1) % _n] = 0 ->
15          fork[(id+1) % _n] <-- 1;
16          status <- 2;
17          fork[(id+1) % _n] <-- 0;
18          status <- 3;
19          fork[id] <-- 0;
20          status <- 0;
21          Behavior
22  }
23
24  check {
25    NoDeadlock =
26      always exists Phil p,
27        status of p != 1
28  }
```

(a) Dining philosophers.

```
 1  system {
 2    extern = _n
 3    spawn = Node: _n
 4  }
 5
 6  stigmergy Election {
 7    link = true
 8    leader: _n
 9  }
10
11  agent Node {
12    stigmergies = Election
13    Behavior =
14      leader > id ->
15        leader <~ id;
16        Behavior
17  }
18
19  check {
20    LeaderIs0 =
21      eventually forall Node a,
22        leader of a = 0
23  }
```

(b) Leader election.

Fig. 1. Two example systems in LAbS.

invariant `NoDeadlock` (lines 25–27) states that the system should never reach a state where all agents are waiting for the second fork.

Figure 1b contains a simple *leader election* system, which we will use to illustrate stigmergic variables. Lines 6–9 define a stigmergy `Election` containing a single variable `leader`. The `link` predicate is, in general, a Boolean expression over the state of two agents: an agent may only send a stigmergic message to another one if they satisfy this predicate. In this case, the predicate is simply `true`, so any two agents may communicate at any time. The stigmergic variable `leader` is initially set to the value of external parameter `_n`. The definition of `Node` agents states that they can access the `Election` stigmergy (line 12). Their behavior (lines 13–16) simply tells them to repeatedly update the variable `leader` to their own `id` as long as it contains a greater value. Finally, property `LeaderIs0` (lines 20–22) specifies that the system should eventually reach a state where all `Node` agents agree on a value of `0` for variable `leader`.

Supported Properties. SLiVER currently supports invariants and inevitable reachability properties. A property is expressed by a modality keyword (`always` for invariants, `eventually` for inevitability properties), followed by a predicate over the state of agents. The predicate may contain existential (`exists`) or universal (`forall`) quantifiers. Alternation of existential and universal quantifiers in the same property is not supported yet.

LNT and CADP. LNT is a formally defined language for the description of asynchronous concurrent systems [17]. A system is modeled as a process, generally composed of several, possibly concurrent processes, which may perform communication actions on gates and exchange information by multiway (value-passing) rendezvous, in the style of the Theoretical CSP [19] and LOTOS [20] process algebras. The syntax of LNT is inspired from both imperative languages (assignments, sequential composition, loops) and functional languages (pattern matching, recursion), with many static checks, such as binding, typing, and dataflow analysis ensuring the proper definition of variables and function results.

CADP [16] is a software toolbox for the analysis of asynchronous concurrent systems, in particular systems described in LNT. It contains a wide range of tools for simulation, test generation, verification (model checking and equivalence checking), performance evaluation, etc. We briefly describe two CADP tools named Evaluator and Executor. Evaluator is a model checker that can evaluate properties expressed in the language MCL [25], a temporal logic based on the modal μ-calculus [21] extended with regular action formulas and value-passing constructs.[3] Executor, on the other hand, performs a bounded random exploration of the state space of a given program. Starting from the initial state, it repeatedly enumerates and then randomly chooses one of the transitions going out of the current state, until it has generated a sequence of the requested length. Explorations can be made reproducible by manually providing a seed for the internal pseudo-random number generator.[4]

3 Overview of SLiVER

Workflow. The analysis workflow is shown in Fig. 2. First, a front end parses the input file and substitutes external parameters with the values provided in the command line, to obtain a system specification $\mathbb{S}$ and a property of interest φ. After that, we perform a two-step encoding procedure. The first step is independent of the target language and builds a structural symbolic representation $\mathbb{T}$ of the behaviors of the agents within $\mathbb{S}$. This representation is used in the second step to encode $\mathbb{S}$ and φ into an LNT program $\mathbb{P}$. At this point, a wrapper invokes a specific program from the CADP toolbox, depending on the analysis task requested by the user. In verification mode, the tool invokes Evaluator to model-check $\mathbb{P}$. If a counterexample is found, a translation module converts it to a LAbS-like syntax and shows it to the user; otherwise, the user is notified that φ holds in $\mathbb{S}$. In simulation mode, instead, we call Executor to obtain one or more random traces of $\mathbb{P}$. Each trace is then translated and shown to the user. Simulation traces will also display a message whenever an invariant is violated or an `eventually` property is satisfied.

[3] See http://cadp.inria.fr/man/evaluator.html and http://cadp.inria.fr/man/mcl.html.

[4] See http://cadp.inria.fr/man/executor.html.

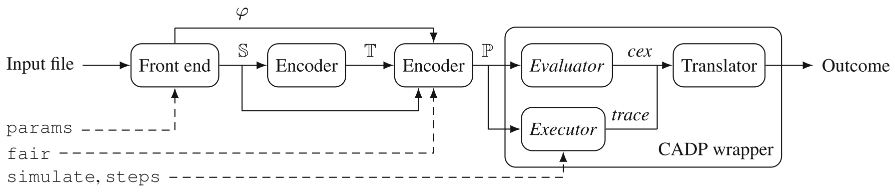

Fig. 2. Workflow of SLiVER with the CADP back end.

Implementation Details and Availability. The front end and encoder are implemented in about 2500 lines of F#, and rely on LNT templates amounting to 450 additional lines. The rest of SLiVER consists of roughly 1000 lines of Python. All Python source code for SLiVER, along with licensing information, is available at https://git.io/sliver-tool. A demonstration video is available at https://drive.google.com/file/d/12kvZXbUiVHRZiXINvOm81D941CYaTeBL.

Usage. This command invokes SLiVER with CADP as the analysis back end:

```
sliver.py <specfile> [params] --backend cadp --fair
          [--simulate <n> --steps <s>]
```

where `specfile` is the name of the input specification file. If the input system is parameterized, the user must provide a sequence `params` in the form `param=val` to assign a value to each parameter. Argument `--backend cadp` is needed to force SLiVER to use the CADP analysis module. As an example, if we invoke SLiVER on the system of Fig. 1a with the command

```
sliver.py philosophers.labs n=5 --backend cadp
```

we obtain the counterexample of Fig. 3a, disproving property `NoDeadlock`.

By default, the tool assumes that there are no constraints on the interleaving of agents. However, in some cases it might be convenient to restrict the analysis to traces where interleaving is restricted according to some policy. Currently, SLiVER allows to enforce round-robin execution of agents through the optional `--fair` flag.

If the optional arguments `--simulate` <n> `--steps` <s> are omitted, the tool attempts to verify the input property on the given system. Otherwise, it returns `n` execution traces, each one containing at most `s` transitions. As an example, Fig. 3b contains part of a simulation trace for the *leader election* system of Fig. 1b, with three agents[5]. This trace shows the asynchronous nature of stigmergic messages. Notice that all stigmergic assignments within the trace show both the value and its attached timestamp. In the first steps, nodes 0 and 2 update `leader` to their respective ids. Then, node 0 sends a confirmation message for `leader`. It does so because it had to compute the guard `leader >`

[5] The full command used to obtain this trace is `sliver.py leader.labs n=3 --backend cadp --simulate 1 --steps 100`.

```
<initialization>
        fork[0] <-- 0
        fork[1] <-- 0
        fork[2] <-- 0
        fork[3] <-- 0
        fork[4] <-- 0
Phil 0: status <- 0
Phil 1: status <- 0
Phil 2: status <- 0
Phil 3: status <- 0
Phil 4: status <- 0
<end initialization>
Phil 0: fork[0] <-- 1
Phil 4: fork[4] <-- 1
Phil 4: status <- 1
Phil 3: fork[3] <-- 1
Phil 3: status <- 1
Phil 2: fork[2] <-- 1
Phil 2: status <- 1
Phil 1: fork[1] <-- 1
Phil 1: status <- 1
Phil 0: status <- 1
<property violated>
```

(a) A counterexample trace for property `NoDeadlock` of Listing 1a.

```
 1  <initialization>
 2  Node 0: leader <~ 3,0
 3  Node 1: leader <~ 3,1
 4  Node 2: leader <~ 3,2
 5  <end initialization>
 6  Node 0: leader <~ 0,3
 7  Node 2: leader <~ 2,4
 8  <Node 0: confirm 'leader'>
 9  Node 1: leader <~ 0,3
10  <Node 0: end confirm 'leader'>
11  <Node 1: propagate 'leader'>
12  <Node 1: end propagate 'leader'>
13  <Node 0: propagate 'leader'>
14  <Node 0: end propagate 'leader'>
15  <Node 2: confirm 'leader'>
16  Node 0: leader <~ 2,4
17  Node 1: leader <~ 2,4
18  <Node 2: end confirm 'leader'>
19  <Node 0: propagate 'leader'>
20  <Node 0: end propagate 'leader'>
21  Node 0: leader <~ 0,8
22  <Node 0: propagate 'leader'>
23  Node 1: leader <~ 0,8
24  Node 2: leader <~ 0,8
25  <Node 0: end propagate 'leader'>
26  <property satisfied>
```

(b) A simulation of the *leader election* system (Listing 1b).

Fig. 3. Example of SLiVER outputs.

`id`. Node 1 picks up the message and updates its value of `leader` accordingly (lines 8–10). On the other hand, node 2 ignores the message, since its own value of `leader` has a higher timestamp. After a sequence of messaging rounds, during which node 0 sets `leader` to 2 (line 16), the same node updates yet again `leader` to 0 (line 21). Then, a propagation messages from node 0 forces the other nodes to accept that value for `leader`, and property `LeaderIs0` becomes satisfied (line 26).

The tool supports other flags, not shown above. If an invocation is enriched with `--verbose`, SLiVER will print the full output from the back end. The `--debug` flag enables the output of additional messages for diagnostic purposes. Finally, the `--show` flag forces SLiVER to print the emulation program and quit without performing any analysis.

4 Program Generation

In this section we describe how we encode a LAbS system $\mathbb{S}$ and a property φ into an LNT *emulation program* $\mathbb{P}$ by using the intermediate representation $\mathbb{T}$.

We illustrate our description with simplified excerpts of LNT code generated from the tool.[6]

Intermediate Representation. The intermediate representation of an agent behavior B contains one record for each basic action within B. Each record is decorated with an *entry condition* and an *exit condition.* An entry condition is a predicate over a set of symbolic variables, which we call the *program counter* of the agent. Intuitively, the program counter tracks the actions which the agent can perform at any given time. An exit condition, on the other hand, is a (possibly nondeterministic) assignment to the program counter. Exit conditions are constructed so as to preserve the control-flow of B. We use multiple variables for the program counter to compactly represent parallel compositions of LAbS processes within a single behavior.

Program Stub. Once the intermediate representation $\mathbb{T}$ is obtained, the generation of the emulation program $\mathbb{P}$ starts from a stub, containing a type definition `Sys` that encodes the full state of $\mathbb{S}$. A system is composed of a collection of `agents`, an environment `env`, and a global clock `time` (Listing 1, lines 1–3). The latter is needed to model the semantics of stigmergic variables. Throughout Listing 1, the `with "get", "set"` construct implements standard functions for accessing and updating elements (for array types) or fields (for record types). The LNT type `Agent` models a LAbS agent: each agent has an identifier `id`, a program counter `pc`, two stores `I` and `L` respectively used for local and stigmergic variables, two stores `Zprop` and `Zconf` to keep track of pending propagation and confirmation messages, and an `init` field that tracks whether the agent has been initialized (lines 4–8). `Agents`, `Env`, `PC`, `Iface`, `Lstig`, and `Pending` are all implemented as arrays (lines 10–12).

Their sizes are determined by SLiVER through static analysis of the input specifications. $\#spawn$ is the total number of agents within the system, as specified in the `spawn` section (e.g., at line 4 in Fig. 1a). $\#\mathcal{I}$, $\#\mathcal{L}$, and $\#\mathcal{E}$ respectively denote the number of internal, stigmergic, and shared variables within the behavioral specifications. $\#\mathcal{P}$ is the number of program counter variables, which is computed during the construction of $\mathbb{T}$. Finally, type `ID` is a natural number strictly less than the number of agents in the system (line 16). The stub also contains LNT functions and processes that implement the semantics of LAbS, and thus never change (see Sect. 4.1 for an example of such a process). Notice that SLiVER is able to alter this stub according to the features of $\mathbb{S}$. For instance, if the system does not feature any stigmergic variables, the emulation program will not contain `Lstig`, `Pending`, nor the functions that implement stigmergic messaging, and the `Sys` type will not have a `time` field.

[6] The full LNT programs for the *dining philosophers* system (with `_n` = 5) and the *leader election* one (with `_n` = 3) can be found at https://git.io/philosophers-lnt and https://git.io/leader-lnt, respectively.

```
type Sys is
  sys(agents: Agents, env: Env, time: Nat) with "get", "set"
end type
type Agent is
  agent(id: ID, I: Iface, pc:PC,
    L: Lstig, Zprop:Pending, Zconf: Pending, init: Bool)
  with "get", "set"
end type

type Agents is array [ 0 .. #spawn−1 ] of Agent with "get", "set" end type
type Env is array [ 0 .. #ℰ − 1 ] of Int with "get", "set" end type
type PC is array [ 0 .. #𝒫 − 1 ] of Nat with "get", "set" end type
type Iface is array [ 0 .. #ℐ − 1 ] of Int with "get", "set" end type
type Lstig is array [ 0 .. #ℒ − 1 ] of Int with "get", "set" end type
type Pending is array [ 0 .. #ℒ − 1 ] of Bool with "get", "set" end type
type ID is X:Nat where X < #spawn end type
```

Listing 1: Type definitions.

Emulation Functions. We populate the stub by encoding each record within $\mathbb{T}$ as a separate LNT process. We call these processes *emulation functions.* An emulation function for a given record alters the state of the system according to the semantic rule of its action, and then updates the program counter of the selected agent according to its exit condition. For instance, Listing 2 emulates action

`fork[id] = 0 -> fork[id] <-- 1`

from the *dining philosophers* example (lines 11–12 of Fig. 1a). The guard is encoded by the `only if ... then ... end if` construct, while the assignment to `fork[id]` is represented by the update of the corresponding element of array `E` (lines 20–21). We refer the reader to Sect. 4.2 for additional examples of emulation functions.

The main section of the program (Listing 3) implements a *scheduler*, that repeatedly selects an agent and calls an emulation function. Agent selection happens by assigning a value to a variable `id`. If the tool is invoked with the `--fair` flag, the variable is simply incremented modulo the number of agents; otherwise, a nondeterministic assignment is performed (lines 34–37). Listing 4 shows the LNT process implementing an iteration of the scheduler. Notice that an emulation function may only be called if the program counter of the selected agent satisfies its corresponding entry condition (see e.g. lines 48–50). This prevents spurious executions. At each iteration, instead of calling an emulation function, the scheduler may call one of several system functions implementing other semantic rules of the language, e.g., communication between agents (line 39).

```
17  process action_0_9
18  (in out a:Agent, in out E:Env)
19  is
20    only if (E[a.id]) == 0) then
21      E[a.id] := 1;
22      var p: PC in
23        p := a.pc; p[0] := 8;
24        agent := a.{pc => p}
25      end var
26    end if
27  end process
```

Listing 2: An emulation function.

```
28  process MAIN [m:Any] is
29    -- initialize sys and id
30    loop
31      monitor[m](sys.agents);
32      select
33        step(?!sys, id);
34        -- if-fair flag was used
35        id := (id + 1) mod #spawn
36        -- otherwise
37        id := any ID
38      []
39        -- system functions
40      end select
41    end loop
42  end process
```

Listing 3: Main section of $\mathbb{P}$.

Property Instrumentation. The generated program is then instrumented for the verification of φ. First, we obtain a propositional formula φ' from φ by quantifier elimination. Then, we add a *monitor* process to $\mathbb{P}$, which is executed before each iteration of the scheduler (Line 23 of Listing 3). A stub of the monitor process is shown in Listing 5. If φ is an invariant and φ' is violated, the monitor emits a *false* value over a gate `m` (line 63). On the other hand, if φ is an inevitable reachability property and φ' holds, a *true* value will be emitted over `m` (line 68). In any case, when the monitor emits a value, it also terminates $\mathbb{P}$ by means of a `stop` instruction, since there is no need to further explore the evolution of $\mathbb{P}$. This instruction is only added to the program when in verification mode: in simulation mode, the program will keep running until it reaches either a deadlocked state or the user-provided bound.

```
43  process step (in out sys:Sys, i:ID)
44  is
45    ...
46    agent := sys.agents[i]; E := sys.env;
47    select
48      only if (agent.pc[0] == 9) then
49        action_0_9(!?agent, !?E)
50      end if
51      []
52      ...
53    end select;
54    agents[i] := agent;
55    sys := sys.{agents => agents, env => E,
56      time => sys.time + 1}
57  end process
```

Listing 4: A scheduler iteration.

```
58  process monitor [m: Any]
59  (sys:Sys)
60  is
61    -- invariants
62    if not(φ′) then
63      m(false);
64      stop
65    end if
66    -- inevitability
67    if φ′ then
68      m(true);
69      stop
70    end if
```

Listing 5: Property encoding.

Size of Emulation Programs. The behavior of multiple identical agents is only encoded once, by parameterizing all emulation functions in the *id* of the agent. Therefore, the number of lines of code in $\mathbb{P}$ scales well with the number of agents in the input system. To show that, we consider the systems of Fig. 1a–1b, as well as the *boids* and *majority* systems introduced in [10]. For each one, we build a 10-agent and a 100-agent emulation program, and compare their sizes. Table 1 shows the size of the input specification and of the two programs. *Dining philosophers* is the only system where the size of $\mathbb{P}$ increases, roughly by a factor of 1.5. This is due to initialization code for array `forks`, whose length depends on the number of agents. The other systems have a fixed-size state, and thus their encodings have the same size, regardless of the number of agents. The growth of the *dining philosophers* program may be avoided by improving the LNT code generator, e.g., by initializing LAbS arrays within a loop. We plan to implement improvements of this kind in a future release of SLiVER.

Table 1. Size of LNT emulation programs with respect to the number n of agents.

Input system		LNT size	
Name	Size	$n = 10$	$n = 100$
Boids	55	530	530
Dining philosophers	28	332	512
Leader election	26	344	344
Majority	57	584	584

```
process propagate (in out sys: Sys) is
  var senderId:ID, key: Nat, sender:Agent, agents:Agents,
  j, k: Nat, L: Lstig, a:Agent in
    senderId := any ID where not(empty(sys.agents[senderId].Zprop));
    agents := sys.agents; sender := agents[senderId];
    key := any Nat where member(key, sender.Zprop);
      for j := 0 while j < #spawn by j := j + 1 loop
        a := agents[j];
        if (a.id != sender.id) and link(sender, a, key) and
        (a.L[key].tstamp < sender.L[key].tstamp) then
          L := a.L;
          for k := key while k <= TUPLEEND(key) by k := k + 1 loop
            L[k] := sender.L[k];
          end loop;
          agents[j] := a.{
            L => L, Zprop => insert(key, a.Zprop),
            Zconf => remove(key, a.Zconf) }
        end if
      end loop;
      agents[senderId] := sender.{Zprop => remove(key, sender.Zprop)};
      sys := sys.{agents => agents}
  end var
end process
```

Listing 6: Propagation of stigmergic variables in LNT.

4.1 Example: A System Function

Listing 6 contains an LNT process that implements LAbS propagation messages. This process may be called at each iteration of the scheduler of the emulation program (line 39 of Listing 3). A similar function, not shown here, implements confirmation messages.

The process first selects an agent with at least one pending message, i.e., with a non-empty `Zprop` field. The selection happens via a nondeterministic assignment of an agent identifier to a variable `senderId` (line 4). Once a suitable sender is found, an element of `Zprop` is nondeterministically selected and stored in the `key` variable (line 6). This value is the index of the stigmergic variable that will be propagated. The process then finds all potential receivers of the message: sender and receiver must be different agents, and they have to satisfy the `link` predicate for the stigmergic variable that is being sent (line 9).

If an agent satisfies all the above requirements, it can receive the message. Furthermore, if its own timestamp for `key` is less than the one of the sender (line 10), it will update its value and timestamp for `key` with the ones from the message (otherwise, it will just discard it). Notice that multiple stigmergic variables may actually be updated (lines 12–14). This is because LAbS allows the user to put multiple stigmergic variables together in a *tuple*, and its semantics guarantee that variables within a tuple are always propagated together [10]. The loop in the LNT process enforces these guarantees. In lines 15–17, the state of the receiver is updated, and `key` is added to its set of pending propagation messages. Additionally, `key` is removed from its pending *confirmation* messages: intuitively, the agent needs no further confirmation for that variable, since it has just received a newer value. Finally, the value `key` is removed from the pending propagation messages of the sender (line 20).

4.2 Example: Emulation Functions

Listing 7 contains all LNT emulation functions for the *dining philosophers* example. The name of each emulation function is constructed from its entry condition. For instance, function `action_0_2` has entry condition `pc[0] == 2`. A comment within each process reports its corresponding LAbS action. Updates to local and shared variables are implemented through the *attr* and *env* processes, respectively. Notice how the assignments to the program counter at the end of each function preserve the control flow of the input specification.

5 Property Verification

In this section we explain how we determine whether a system $\mathbb{S}$ satisfies a property φ by model-checking the emulation program generated from $(\mathbb{S}, \varphi)$. We use the Evaluator tool to verify the values emitted by the `monitor` process (Listing 5). If φ is an invariant, we check that the program never emits a *false* value over `m`. This property is encoded as the MCL query

```
[ true * . "M !FALSE"]false
```

```
process action_0_2
(in out a: Agent, in out E: Env) is
--status <- 0
 attr(!?a, 0, 0);
 var p: PC in
  p := a.pc; p[0] := 9;
  a := a.{pc => p}
 end var
end process

process action_0_3
(in out a: Agent, in out E: Env) is
--fork[id] <-- 0
 env(!?E, 0 + (a.id), 0);
 var p: PC in
  p := a.pc; p[0] := 2;
  a := a.{pc => p}
 end var
end process

process action_0_4
(in out a: Agent, in out E: Env) is
--status <- 3
 attr(!?a, 0, 3);
 var p: PC in
  p := a.pc; p[0] := 3;
  a := a.{pc => p}
 end var
end process

process action_0_5
(in out a: Agent, in out E: Env) is
--fork[id + 1 % 5] <-- 0
 env(!?E, 0 + (a.id + 1) mod 5, 0);
 var p: PC in
  p := a.pc; p[0] := 4;
  a := a.{pc => p}
 end var
end process

process action_0_6
(in out a: Agent, in out E: Env) is
--status <- 2
 attr(!?a, 0, 2);
 var p: PC in
  p := a.pc; p[0] := 5;
  a := a.{pc => p}
 end var
end process

process action_0_7
(in out a: Agent, in out E: Env) is
--(fork[id + 1 % 5]) == (0)->
--fork[id + 1 % 5] <-- 1
 only if (E[(a.id + 1) mod 5]) == 0)
    then
  env(!?E, 0 + (a.id + 1) mod 5, 1);
  var p: PC in
   p := a.pc; p[0] := 6;
   a := a.{pc => p}
  end var
 end if
end process

process action_0_8
(in out a: Agent, in out E: Env) is
--status <- 1
 attr(!?a, 0, 1);
 var p: PC in
  p := a.pc; p[0] := 7;
  a := a.{pc => p}
 end var
end process

process action_0_9
(in out a: Agent, in out E: Env) is
--(fork[id]) == (0)->fork[id] <-- 1
 only if (E[a.id]) == 0) then
  env(!?E, 0 + (a.id), 1);
  var p: PC in
   p := a.pc; p[0] := 8;
   a := a.{pc => p}
  end var
 end if
end process
```

Listing 7: Emulation functions for the dining philosopherssystem.

When φ is an inevitability property, instead, we check that all *fair executions* [27] of $\mathbb{P}$ emit a value of *true* over `m` at some point. To do that, we use the following MCL query:

```
[ (not ("M !TRUE"))* ]<true * . "M !TRUE">true
```

To trust that the outcome of the model checker is also a verdict on the original problem (namely, whether φ holds in $\mathbb{S}$), we need to prove that intermediate representation $\mathbb{T}$ preserves all traces of each behavior in the system, and also that the emulation program $\mathbb{P}$ correctly interleaves these traces with calls to system functions, without introducing spurious executions. We cannot include a detailed proof for reasons of space, but this procedure adapts a previous structure-aware encoding [11] (which was tied to explicit-state model checking) to the semantics of LAbS, and makes it independent of the verification technique. Thus, our argument for correctness closely follows the one for that encoding.

6 Related Work

There are several specialized tools for the formal analysis of multi-agent systems. MCMAS [24] verifies multi-agent systems of unbounded size with synchronous communication. Its language lacks value-passing actions, so it is not clear whether their technique could be applied to LAbS. AJPF [7] can perform explicit-state model-checking on a variety of agent-oriented languages. Differently from AJPF, SLiVER is modular with respect to the analysis back end, and may support explicit-state techniques as well as symbolic ones, such as SAT-based bounded model checking [10]. Peregrine [6] can verify and simulate *population protocols*, i.e. collections of identical mobile agents [2]. It can check that a population of unbounded size inevitably ends up satisfying a given predicate over its initial state. SLiVER cannot reason over unbounded-size systems, but it allows for the verification of invariants in addition to inevitable reachability properties.

The concept of verifying domain-specific languages by means of a structural translation into more amenable formalisms is not new. For instance, in [18] hardware specifications are translated into LOTOS and verified with CADP, while [11] shows a translation from an attribute-based process algebra [1] to UMC [30].

7 Conclusion

We have presented an automated analysis workflow for multi-agent systems based on CADP and implemented as part of the SLiVER tool. Through an LNT encoding, the workflow allows to formally verify the input system via model checking, as well as generate random execution traces. The end user does not need to be familiar with either LNT or CADP: knowledge of the input language LAbS is the only requirement.

Future work may improve the presented workflow at several levels. We currently represent the whole system as a *sequential* LNT program: one might instead represent agents as parallel processes and apply compositional verification [15,22,23] to improve model checking performance. We could verify much more expressive properties than the current ones, by devising a translation into MCL queries with data variables [25] to be passed to the model checker. This would require an extension of the property language currently understood by the tool, as well as a correct encoding of this (state-based) language into MCL, which is action-based [12]. Finally, we could use the new trace generation capability to implement simulation-based analysis techniques, such as statistical model checking [28].

References

1. Abd Alrahman, Y., De Nicola, R., Loreti, M.: A calculus for collective-adaptive systems and its behavioural theory. Inf. Comput. **268** (2019). https://doi.org/10.1016/j.ic.2019.104457

2. Aspnes, J., Ruppert, E.: An introduction to population protocols. In: Garbinato, B., Miranda, H., Rodrigues, L. (eds.) Middleware for Network Eccentric and Mobile Applications, pp. 97–120. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-89707-1_5
3. Axtell, R.L., et al.: Population growth and collapse in a multiagent model of the Kayenta Anasazi in Long House Valley. Proc. Natl. Acad. Sci. **99**(suppl 3), 7275–7279 (2002). https://doi.org/10.1073/pnas.092080799
4. Baeza, A., Janssen, M.A.: Modeling the decline of labor-sharing in the semi-desert region of Chile. Reg. Environ. Change **18**(4), 1161–1172 (2017). https://doi.org/10.1007/s10113-017-1243-0
5. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
6. Blondin, M., Esparza, J., Jaax, S.: Peregrine: a tool for the analysis of population protocols. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 604–611. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_34
7. Bordini, R.H., Dennis, L.A., Farwer, B., Fisher, M.: Automated verification of multi-agent programs. In: 23rd International Conference on Automated Software Engineering (ASE), pp. 69–78. IEEE (2008). https://doi.org/10.1109/ASE.2008.17
8. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.: Synthesising interprocedural bit-precise termination proofs. In: 30th International Conference on Automated Software Engineering (ASE), pp. 53–64. IEEE (2015). https://doi.org/10.1109/ASE.2015.10
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
10. De Nicola, R., Di Stefano, L., Inverso, O.: Multi-agent systems with virtual stigmergy. Sci. Comput. Program. **187** (2020). https://doi.org/10.1016/j.scico.2019.102345
11. De Nicola, R., Duong, T., Inverso, O., Mazzanti, F.: Verifying properties of systems relying on attribute-based communication. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd. LNCS, vol. 10500, pp. 169–190. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_9
12. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-53479-2_17
13. Farmer, J.D., Foley, D.: The economy needs agent-based modelling. Nature **460**(7256), 685–686 (2009). https://doi.org/10.1038/460685a
14. Gadelha, M.Y.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: an industrial-strength C model checker. In: 33rd International Conference on Automated Software Engineering (ASE), pp. 888–891. ACM (2018). https://doi.org/10.1145/3238147.3240481
15. Garavel, H., Lang, F., Mateescu, R.: Compositional verification of asynchronous concurrent systems using CADP. Acta Informatica **52**(4–5), 337–392 (2015). https://doi.org/10.1007/s00236-015-0226-1
16. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Softw. Tools Technol. Transf. **15**(2), 89–107 (2013). https://doi.org/10.1007/s10009-012-0244-z

17. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd. LNCS, vol. 10500, pp. 3–26. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_1
18. Garavel, H., Salaün, G., Serwe, W.: On the semantics of communicating hardware processes and their translation into LOTOS for the verification of asynchronous circuits with CADP. Sci. Comput. Program. **74**(3), 100–127 (2009). https://doi.org/10.1016/j.scico.2008.09.011
19. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
20. ISO/IEC: LOTOS – A formal description technique based on the temporal ordering of observational behaviour. International Standard 8807 (1989)
21. Kozen, D.: Results on the propositional μ-Calculus. Theoret. Comput. Sci. **27**, 333–354 (1983). https://doi.org/10.1016/0304-3975(82)90125-6
22. Lang, F., Mateescu, R., Mazzanti, F.: Compositional verification of concurrent systems by combining bisimulations. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 196–213. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_13
23. Lang, F., Mateescu, R., Mazzanti, F.: Sharp congruences adequate with temporal logics combining weak and strong modalities. TACAS 2020. LNCS, vol. 12079, pp. 57–76. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_4
24. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. Int. J. Softw. Tools Technol. Transf. **19**(1), 9–30 (2015). https://doi.org/10.1007/s10009-015-0378-x
25. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_12
26. Pinciroli, C., Lee-Brown, A., Beltrame, G.: A tuple space for data sharing in robot swarms. In: 9th International Conference on Bio-inspired Information and Communications Technologies (BICT), pp. 287–294. ICST/ACM (2015). https://doi.org/10.4108/eai.3-12-2015.2262503
27. Queille, J.P., Sifakis, J.: Fairness and related properties in transition systems - a temporal logic to deal with fairness. Acta Informatica **19**, 195–220 (1983). https://doi.org/10.1007/BF00265555
28. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_16
29. Stiglitz, J.E., Gallegati, M.: Heterogeneous interacting agent models for understanding monetary economies. Eastern Econ. J. **37**(1), 6–12 (2011). https://doi.org/10.1057/eej.2010.33
30. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Sci. Comput. Program. **76**(2), 119–135 (2011). https://doi.org/10.1016/j.scico.2010.07.002
31. Winikoff, M.: Assurance of agent systems: what role should formal verification play? In: Dastani, M., Hindriks, K., Meyer, J.J. (eds.) Specification and Verification of Multi-Agent Systems, pp. 353–383. Springer, Heidelberg (2010). https://doi.org/10.1007/978-1-4419-6984-2_12