Verification of Distributed Systems via Sequential Emulation

Luca Di Stefano, Rocco de Nicola, Omar Inverso

Verification of Distributed Systems via Sequential Emulation

LUCA DI STEFANO, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, France
ROCCO DE NICOLA, IMT School of Advanced Studies, Italy
OMAR INVERSO, Gran Sasso Science Institute (GSSI), Italy

Sequential emulation is a semantics-based technique to automatically reduce property checking of distributed systems to the analysis of sequential programs. An automated procedure takes as input a formal specification of a distributed system, a property of interest and the structural operational semantics of the specification language and generates a sequential program whose execution traces emulate the possible evolutions of the considered system. The problem as to whether the property of interest holds for the system can then be expressed either as a reachability or as a termination query on the program. This allows to immediately adapt mature verification techniques developed for general-purpose languages to domain-specific languages, and to effortlessly integrate new techniques as soon as they become available. We test our approach on a selection of concurrent systems originated from different contexts from population protocols to models of flocking behaviour. By combining a comprehensive range of program verification techniques, from traditional symbolic execution to modern inductive-based methods such as property-directed reachability, we are able to draw consistent and correct verification verdicts for the considered systems.

CCS Concepts: • **General and reference → Verification**; • **Software and its engineering → Automated static analysis**; • **Theory of computation → Process calculi**.

Additional Key Words and Phrases: Concurrency, Distribution, Process Algebra, Structural Operational Semantics, Semantics-based Verification, Domain-specific Languages, Program Verification, Reachability, Termination, Sequentialization.

ACM Reference Format:
Luca Di Stefano, Rocco De Nicola, and Omar Inverso. 2021. Verification of Distributed Systems via Sequential Emulation. *ACM Trans. Softw. Eng. Methodol.* xx, yy, Article zz (December 2021), 41 pages.

1 INTRODUCTION

In some natural and artificial systems, the intricacies of concurrency manifest themselves in the subtle relationship between the simple behaviour of the agents and the unexpected emerging consequences of their mutual interference [70].

To experience this first-hand, one can observe long enough a colony of ants, or the evolutions of a social network, or many other seemingly different systems such as markets, populations, robotic swarms, as well as several classes of so-called complex or collective systems [13, 28, 42, 46, 99, 101].

From a verification perspective, systems of this kind can be particularly difficult to deal with. At a specification level, their specific characteristics may not be easily expressible through the usual

*Institute of Engineering Univ. Grenoble Alpes

Author's addresses: Luca Di Stefano, Inria, 38334 Montbonnot Saint-Martin, France, luca.di-stefano@inria.fr; Rocco De Nicola, IMT School of Advanced Studies, 55100 Lucca, Italy, rocco.denicola@imtlucca.it; Omar Inverso, Gran Sasso Science Institute (GSSI), 67100 L'Aquila, Italy, omar.inverso@gssi.it.

1049-331X/2021/12-ARTzz $15.00
https://doi.org/

constructs of a general-purpose formalism. As for the analysis, one can expect remarkably large state spaces caused by process interleaving and asynchrony.

Appropriate domain-specific languages, based on bottom-up paradigms such as agents [71], actors [48], and process calculi [34], and tailored analysis techniques, can tackle these limitations [12, 68, 80, 97]. However, they do entail a non-negligible integration effort on a case-by-case basis. Moreover, they make it difficult to catch up with the most recent advances in automated analysis proposed for mainstream languages. This prevents opening up towards other disciplines where there might be a great deal to be gained.

To reduce this distance, in this paper we present *sequential emulation*, a semantics-based reduction from property checking of concurrent system specifications to the analysis of sequential imperative programs. A two-step encoding procedure takes as input the formal specifications of the system under consideration, the property of interest, and the operational semantics of the domain-specific language in which the system is described. The formal specifications define the behaviour of the different agents in the systems as separate processes. The structural operational semantics (SOS) rules [83] of the language describe the effect of the actions of the agents on themselves and on the rest of the system, and the possible interactions.

The formal specifications are first translated into a compact representation of the possible flow of actions of each agent and of the interleaving of the actions of the different agents. A *symbolic* control mechanism based on predicates avoids explicitly representing the interleaving of the actions and thus enables it to retain the compactness of the initial specifications. The representation resulting from the first translation is then combined with the SOS rules of the specification language to generate a simple *sequential* program where separate functions model the actions of the agents and the possible system-level transitions; a *scheduler* repeatedly invokes such emulation functions, non-deterministically, to model the additional interleaving of the actions of the agents with the system-level transitions. Compactness is still retained as the program size is linear in the overall number of elementary actions used in the specifications. By construction, the execution traces of the sequential program emulate the possible evolutions of the distributed system under consideration. Therefore, depending on the property of interest, the program can be instrumented either for reachability or termination analysis, thus reducing formal verification of the initial system to a common verification query on a sequential program.

Our approach can in principle be adapted by mapping from any domain-specific language equipped with an SOS semantics to any imperative language with arrays and loops. The adaptation only requires a one-time manual effort by the user, who has to provide templates for the emulation functions in the target language. All available verification techniques which target this imperative language and support standard instrumentation constructs for automated analysis (i.e., non-deterministic initialisation, assertions, assumptions) can then simply be used as a black box. As an immediate advantage, we leverage a comprehensive range of readily available techniques for the analysis of sequential programs. Furthermore, we can easily integrate new techniques as soon as they become available. At the same time, the end-user is still at ease, as familiarity with the source language is the only usage requirement.

To evaluate our technique, we consider a selection of distributed systems, namely flocks [89], population protocols [3], and pattern-forming agents [98]. As they originate from different contexts, these systems are normally studied separately and with rather different techniques. In contrast, we express their specifications into the same formal specification language [26]. We then use our approach to automatically translate the specifications into sequential C programs, thus immediately inheriting a comprehensive range of mature verification techniques, including symbolic execution [55], bounded model checking [11], value- and predicate-based abstraction [8], k-induction [95],

inductive strengthening [16, 32, 57], and interprocedural analysis [88]. We can thus make different attempts to analyse each system using every technique.

For all the considered systems, at least one of the used tools does generate a conclusive verification verdict, and the verdicts are always consistent. Therefore, we can confidently draw safe conclusions for every system. We also manage to successfully verify a particularly complex model of flocking behaviour [89] for which, to the best of our knowledge, the only known attempts at formal analysis are limited to simulation and under-approximation. Finally, to illustrate another potential application of our procedure, we briefly consider the domain of service choreographies, and show that we can use sequential emulation to check that a choreography implementation is deadlock-free.

To summarise, our contributions are:

(1) a technique for translating from a given formal language for concurrent systems to a given sequential language for imperative programs;
(2) a prototype implementation of our encoding from a domain-specific language to the C language;
(3) an experimental evaluation on a variety of systems originated from different contexts using a representative set of state-of-the-art verification tools for sequential C programs;
(4) a preliminary demonstration of the applicability of our encoding in the domain of service choreographies.

The rest of the paper is structured as follows. Section 2 introduces some preliminary concepts, as well as the domain-specific language used to illustrate our procedure. Section 3 presents our technique. Section 4 describes the prototype implementation for the aforementioned language, and the experimental results. Section 5 introduces the concept of service choreographies and shows how we can check deadlock freedom of choreography implementation by applying our technique. Finally, Sections 6 and 7 discuss related work and report some concluding remarks.

2 BACKGROUND

In this section, to make the paper self contained, we introduce some basic definitions and concepts that will be used throughout the paper. First, we will introduce the basic notions of process algebras [25] and define labelled transition systems (definition 2.1), provide an overview of Milner's Calculus of Communicating Systems (definition 2.2) and introduce synchronisation algebras (definition 2.3) that are useful to model systems behaviours. After this, we will provide a minimal description of the Linear Temporal Logic formalism (definition 2.5) that is instead useful to model system properties. Finally, we briefly describe a domain-specific language for multi-agent systems, which we will use to demonstrate the applicability of our approach.

2.1 Process Algebras

An important component of our approach is *operational semantics*, which is used to model a program as a labelled transition system (LTS) that consists of a set of states, a set of transition labels, and a transition relation. The states of the transition system are just terms of a language, while the labels of the transitions between states represent the actions or the interactions that are possible from a given state and the state that is reached after the action is performed.

Definition 2.1 (Labelled Transition Systems). A labelled transition system (LTS) is a triple $\langle S, \Lambda, \rightarrow \rangle$, with S representing a set of *states*, L a set of *labels*, and $\rightarrow \subseteq S \times L \times S$ a *labelled transition relation*. Each element of $\rightarrow$ is called a *transition* and is commonly written as $s \xrightarrow{\alpha} s'$, indicating that the LTS may evolve from some state s to a state s' by performing an action with label α.

Table 1. Structural operational semantics of CCS. The rules symmetrical to (CHOICE) and (PAR_1) have been omitted for brevity.

$$\frac{}{\mu.P \xrightarrow{\mu} P}\ (\text{ACT}) \qquad \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}\ (\text{CHOICE})$$

$$\frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}\ (\text{PAR}_1) \qquad \frac{P \xrightarrow{\mu} P' \quad Q \xrightarrow{\overline{\mu}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ (\text{PAR}_2)$$

$$\frac{P \xrightarrow{\mu} P' \quad \mu, \overline{\mu} \notin \Theta}{P \setminus \Theta \xrightarrow{\mu} P' \setminus \Theta}\ (\text{RES}) \qquad \frac{P \xrightarrow{\mu} P'}{P[f] \xrightarrow{f(\mu)} P'[f]}\ (\text{REL}) \qquad \frac{P \xrightarrow{\mu} P' \quad K \triangleq P}{K \xrightarrow{\mu} P'}\ (\text{CON})$$

To provide examples of the actual transformations induced by our approach we will use specifications written in CCS, the Calculus of Communicating Systems [72, 73] introduced by Robin Milner around 1980. Its labels model actions and binary communications, while the set of operators includes primitives for describing parallel composition, choice between actions, and scope restriction. Its operational semantics associates an LTS to each CCS term.

Definition 2.2 (Calculus of Communicating Systems). Let $\Delta = \{a, b, \ldots\}$ a fixed set of *names*, and $\overline{\Delta} = \{\overline{x} \mid x \in \Delta\}$ a set of *co-names*. Let τ be a distinguished *invisible action*. Then, $\Lambda = \Delta \cup \overline{\Delta}$ is the set of *visible actions*, $\Lambda_\tau = \Lambda \cup \tau$ is the set of *generic actions*, and a CCS *process* is any term P generated from the following grammar:

$$P, Q \coloneqq 0 \;\Big|\; \mu.P \;\Big|\; P + Q \;\Big|\; P \mid Q \;\Big|\; P \setminus \Theta \;\Big|\; P[f] \;\Big|\; K$$

where $\mu \in \Lambda \cup \{\tau\}$, Θ is a subset of Δ, and f is a total function $f : \Lambda_\tau \to \Lambda_\tau$ which preserves name/co-name relations (i.e., $f(\overline{a}) = \overline{f(a)}$ for any action a) and does not renames the invisible action (i.e., $f(\tau) = \tau$).

The formal structural operational semantics (SOS) of CCS operators is inductively specified through a set of derivation rules: for each operator there are some rules describing the behaviour of a system in terms of the behaviours of its components. As a result, each process term is seen as a component that can interact with other components or with the external environment. The actual rules are reported in Table 1 and enable us to associate to each CCS term an LTS whose transition relation is the least relation that satisfies those rules.

Intuitively, 0 is the *idle* process, which cannot perform any action; $\mu.P$ can perform action μ and continue as P (rule ACT); $P + Q$ can behave as either P or Q (rule CHOICE); $P \mid Q$ alternates the executions of both P and Q (rule PAR_1). Furthermore, P and Q can evolve together when they are willing to perform two *complementary* actions $\mu, \overline{\mu}$ (rule PAR_2). A *restricted* process $P \setminus \Theta$ has the same behaviour as P, but may only perform those actions that do not belong to Θ (rule RES). The *relabelling* operator $P[f]$ renames the actions performed by P according to the relabelling function f (rule REL). Lastly, we assume that there exists a set of *constants* $K \triangleq P$, and that a CCS term may contain references to these constants (rule CON). This allows the grammar to describe infinite behaviours by means of recursive terms: for instance, a term $K \triangleq a.b.K$ repeatedly performs an action a followed by b.

To conclude our overview of CCS, we use it to specify a simplified version of a *two-phase commit* scenario [41] that will be used later in our examples.

Example 2.1 (Two-phase commit). A two-phase commit (2PC) protocol involves a number of *workers*, which must collectively decide whether to commit or rollback a transaction by interacting through a *coordinator*. In the first phase, the coordinator asks the workers to cast a vote. Each worker may either agree or disagree on committing the transaction. In the second phase, if all workers have agreed, the coordinator tells them to commit the transaction; otherwise, the coordinator sends them a rollback request. In any case, the workers send an acknowledgement message back to the coordinator and finalise the transaction.

A system composed of one coordinator and one worker may be described as the parallel composition of two CCS processes:

$$\begin{aligned}
\textsc{Coord} &\triangleq \mathit{vote}.\left(\overline{\mathit{agree}}.\mathit{commit}.\overline{\mathit{commit}}.\mathit{ok}.0 + \overline{\mathit{disagree}}.\mathit{rollback}.\overline{\mathit{rollback}}.\mathit{nok}.0\right)\\
\textsc{Worker} &\triangleq \overline{\mathit{vote}}.\mathit{agree}.\left(\overline{\mathit{commit}}.\mathit{commit}.\textsc{Worker} + \overline{\mathit{rollback}}.\mathit{rollback}.\textsc{Worker}\right)\\
\mathrm{2PC}_{CCS} &\triangleq \textsc{Coord} \mid \textsc{Worker}\\
\mathrm{2PC}'_{CCS} &\triangleq \mathrm{2PC}_{CCS} \setminus \{\mathit{agree}, \mathit{commit}, \mathit{disagree}, \mathit{rollback}, \mathit{vote}\}
\end{aligned}$$

Please consider the way the coordinator performs an action *ok* or *nok* to indicate whether the transaction was finalized or rolled back. Since the Worker process describes a worker that always agrees to commit the transaction, we would expect the *nok* action to be unreachable. However, $\mathrm{2PC}_{CCS}$ does not satisfy this expectation. As the two processes are not *forced* to synchronise, they might simply perform their actions independently from each other, allowing *nok* to be performed. Thus, one needs to introduce a restriction operator (process $\mathrm{2PC}'_{CCS}$) so that Worker and Coord are forced to synchronise on all actions except *ok* and *nok*. This makes the *nok* action unreachable.

The binary communication through complementary actions used in CCS is just one of the possible interaction strategies used in the literature to model interaction between components. Indeed, we might have synchronisation among similar actions rather than complementary ones, but also one-to-many or many-to-many communications instead of one-to-one. So-called *synchronisation algebras* [105] have been introduced to provide a parametric definition which can be instantiated to capture many of the synchronisation and communication strategies presented in the literature.

Definition 2.3 (Synchronisation algebras). Let A be the set of actions that a process may perform. A synchronisation algebra is a symmetric[1] partial function $\sigma : A \times A \cup \{*\} \hookrightarrow A$, where $*$ is a distinguished symbol representing idleness. The behaviour of the parallel composition of two processes P, Q according to a synchronisation algebra σ, which we denote as $P \mid_\sigma Q$, is formalised by the following semantic rules. Note that the rule symmetrical to IDLE has been omitted for brevity.

$$\frac{P \xrightarrow{\mu} P' \quad \sigma(\mu, *) \neq \bot}{P \mid_\sigma Q \xrightarrow{\sigma(\mu,*)} P' \mid_\sigma Q}\ (\textsc{idle}) \qquad \frac{P \xrightarrow{\mu} P' \quad Q \xrightarrow{\mu'} Q' \quad \sigma(\mu, \mu') \neq \bot}{P \mid_\sigma Q \xrightarrow{\sigma(\mu,\mu')} P' \mid_\sigma Q'}\ (\textsc{sync})$$

Rule IDLE says that, if P is willing to become P' by performing an action μ and $\sigma(\mu, *)$ is an action, then the parallel composition may perform $\sigma(\mu, *)$ and become $P' \mid_\sigma Q$. Rule SYNC states that the two processes may synchronise when they are willing to perform actions μ and μ', respectively, and $\sigma(\mu, \mu')$ is an action.

[1] A function is symmetric if its value is the same for all permutations of its arguments.

Table 2. SOS of Hoare's parallel composition operator. The symmetric rule of (IDLE_L) is omitted.

$$\frac{P \xrightarrow{a} P' \quad a \notin L}{P \,|[L]|\, Q \xrightarrow{a} P' \,|[L]|\, Q} \ (\text{IDLE}_\text{L}) \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q' \quad a \in L}{P \,|[L]|\, Q \xrightarrow{a} P' \,|[L]|\, Q'} \ (\text{SYNC}_\text{L})$$

By relying on a specific synchronisation schema we can easily model the parallel composition of CCS. It would be captured by an algebra where

(1) $\sigma(a, b) = \tau$ iff $b = \overline{a}$, $\perp$ otherwise;
(2) $\sigma(a, *) = a$.

This expresses that (1) two agents may synchronise only by performing two complementary actions, and (2) a single agent may interleave its actions with those of others running in parallel at any time.

As another example, let us consider Hoare's parallel composition operator, which is featured in the CSP process algebra [50] and allows for *multi-party synchronisation* between parallel processes. This operator is denoted by $P \,|[L]|\, Q$, where L is a set of actions. Its behaviour is formalised by the semantic rules in Table 2. Intuitively, processes P and Q may interleave the execution of all actions that are not in L (rule IDLE_L), and they are forced to synchronise on those in L (rule SYNC_L). Whenever they do synchronise on an action $a \in L$, the parallel composition performs a transition with label a. This allows an external process to synchronise with the a action again. For instance, a process $a \,|[\{a\}]|\, a \,|[\{a\}]|\, a$ may perform a single a-transition and terminate. The actual synchronisation algebra σ_L for Hoare's parallel composition $\cdot \,|[L]|\, \cdot$ is the following:

(1) $\sigma_L(a, a) = a$ if $a \in L$, $\perp$ otherwise;
(2) $\sigma_L(a, *) = a$ if $a \notin L$, $\perp$ otherwise.

2.2 Linear Temporal Logic

The properties of the systems we will specify are expressed as formulae of a classical temporal logic, namely LTL [84], that are typically interpreted over a variant of labelled transition systems that associate *labels* to states rather than transitions. Such labels indicate the set of *atomic propositions*, i.e., elementary facts about the system, that hold in each state. In this subsection, we introduce such a variant and then use it as a model for LTL formulae.

Definition 2.4 (Kripke structures). Let AP be a set of *atomic propositions*. Then, a Kripke structure is a tuple $\langle S, I, R, L\rangle$ where S is a finite set of *states*, $I \subseteq S$ a set of *initial states*, $R \subseteq S \times S$ a *transition relation*, and $L : S \to \mathcal{P}(AP)$ a *labelling function*.

Given two states s, s', we say that s' is a *successor* of s if sRs'. This means that the system can evolve from state s to s' in one instant. Thus, the transition relation captures the temporal relations between states; it is generally assumed that every state has at least one successor. Given a state s_0, a *path* rooted in s_0 is a sequence of states $\pi = \langle s_0, s_1, \ldots\rangle$ such that $s_i R s_{i+1}$ for every i. Each path π has an associated *trace* $\sigma = \langle L(s_0), L(s_1), \ldots\rangle$, obtained as the sequence of labels associated to each state in π.

Definition 2.5 (Linear Temporal Logic). Let AP be a finite set of *atomic propositions*. Then, an LTL *formula* is any term generated according to the following grammar:

$$\phi \coloneqq \mathit{true} \mid \neg\phi \mid \phi \wedge \phi \mid a \mid \bigcirc\phi \mid \phi \,\mathcal{U}\, \phi \quad \text{where } a \in AP.$$

The meaning of an LTL formula is formally defined by a *satisfaction relation* $\models$, shown in Table 3. There, we denote by σ_i the i-th element, or *step*, of a trace σ, and by σ^i the *suffix* of σ that starts from σ_i. We now informally describe this relation.

Table 3. Formal semantics of LTL.

$$\begin{array}{lll} \sigma \models \mathit{true} & & \sigma \models a \iff a \in \sigma_0 \\ \sigma \models \neg\phi \iff \sigma \not\models \phi & & \sigma \models \bigcirc\phi \iff \sigma^1 \models \phi \\ \sigma \models \phi_1 \wedge \phi_2 \iff \sigma \models \phi_1 \text{ and } \sigma \models \phi_2 & & \sigma \models \phi_1 \,\mathcal{U}\, \phi_2 \iff \exists i \geq 0.\sigma^i \models \phi_2 \text{ and} \\ & & \quad \forall j.0 \leq j < i \Rightarrow \sigma^j \models \phi_1 \end{array}$$

The *true* formula is always satisfied. Boolean connectives work as usual: $\neg\phi$ is satisfied if ϕ is not, while $\phi_1 \wedge \phi_2$ is satisfied if both sub-formulas hold. Other connectives, such as disjunction $\vee$, implication $\rightarrow$, and so on, may be derived from negation and conjunction. Intuitively, a trace satisfies a formula a if the proposition a holds in the first step of the trace; it satisfies $\bigcirc\phi$ ("next ϕ") if ϕ holds from the second step onwards; lastly, it satisfies $\phi_1 \,\mathcal{U}\, \phi_2$ ("ϕ_1 until ϕ_2") if ϕ_2 holds in the i-th step of the trace (for some i) and ϕ_1 holds in all steps before that. We are chiefly interested in two additional modalities, namely $\Diamond\phi$ ("eventually ϕ") and $\Box\phi$ ("always ϕ"). A trace satisfies the former if some step satisfies ϕ, and the latter if all steps satisfy ϕ. These modalities are not usually included in the core grammar of LTL because they can be expressed as $\mathit{true}\,\mathcal{U}\,\phi$ and $\neg\Diamond\neg\phi$, respectively. A system satisfies an LTL formula if all its traces satisfy it.

Example 2.2 (LTL properties for $2PC_{CCS}$). Let us now describe some properties of interest for the 2PC system of Example 2.1, and show how they can be expressed in LTL. For these properties, we use an atomic proposition a for each action, with the simple interpretation that proposition a holds in a given state if that state may perform an a-transition.

An invariant of interest could be "the action *nok* may never be performed". This property is simply encoded as $\Box\neg\mathit{nok}$. Similarly, the property "there is never an *ok* before an *agree*" would be rendered as $\neg\mathit{ok}\,\mathcal{U}\,\mathit{agree}$. Lastly, we could render the property "If there is an *agree*, then there will eventually be an *ok*" as $\Box(\mathit{agree} \rightarrow \Diamond\mathit{ok})$.

2.3 LAbS, a Domain-specific Language for Stigmergic Interaction

As one of our case studies, we adopt an existing formal specification language for multi-agent systems [26]. We chose this language as it can express a representative selection of systems, such as flocks [89], population protocols [3], and pattern-forming agents [98].

A central concept in the considered specification language, LAbS, is *stigmergic interaction*, which is particularly appropriate to the context of collective systems that may exhibit emerging behaviour [49, 81, 100]. Intuitively, a stigmergy is an interaction mechanism based on signs that agents leave during their activity, and that influence the behaviour of the others. This is recurringly observed in nature, e.g., in ant colonies [79]. Agents in LAbS do not directly communicate with each other: rather, they manipulate *stigmergic* variables which mimic such mechanism. Agents may also interact through an *environment*, which is a set of shared variables. The behaviour (or process) of an agent is a composition of basic actions, which are assignments to local, stigmergic, or shared variables. Composite behaviours are obtained via classical process-algebraic operators, namely sequential composition ($P; Q$), nondeterministic choice ($P + Q$), and interleaving ($P \mid Q$). Moreover, a process may be guarded by a predicate over the state of the agent ($g \rightarrow P$).

Rather than providing a full description of the language, here we only describe the stigmergic interaction mechanism and basic assignments to either the stigmergy or the environment, to show how our encoding can cope well with more complex semantics.

The values of the stigmergic variables are asynchronously *propagated* from one agent to another after an assignment, and agents that read a value from a stigmergic variable do ask the others

Table 4. SOS rules for stigmergic interaction in LAbS.

$$\frac{x \in Zp \quad L(x) = (v,t)}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\text{put}(I,L,x,v,t)} \langle I, L, P, Zc, Zp \setminus \{x\}\rangle} \text{(PROPAGATE)}$$

$$\frac{S \xrightarrow{\text{put}(I',L',x,v,t)} S' \quad I', L', I, L \models \psi_x \quad L(x) = (v',t') \quad t' < t}{S \parallel \langle I, L, P, Zc, Zp\rangle \xrightarrow{\text{put}(I',L',x,v,t)} S' \parallel \langle I, L[x \mapsto (v,t)], P, Zc \setminus \{x\}, Zp \cup \{x\}\rangle} \text{(PUT)}$$

$$\frac{x \in Zc \quad L(x) = (v,t)}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\text{qry}(I,L,x,v,t)} \langle I, L, P, Zc \setminus \{x\}, Zp\rangle} \text{(CONFIRM)}$$

$$\frac{S \xrightarrow{\text{qry}(I',L'\,x,v,t)} S' \quad I', L', I, L \models \psi_x \quad time(L,x) < t}{S \parallel \langle I, L, P, Zc, Zp\rangle \xrightarrow{\text{qry}(I',L',x,v,t)} S' \parallel \langle I, L[x \mapsto (v,t)], P, Zc \setminus \{x\}, Zp \cup \{x\}\rangle} \text{(QRY}_1\text{)}$$

$$\frac{S \xrightarrow{\text{qry}(I',L',x,v,t)} S' \quad I', L', I, L \models \psi_x \quad time(L,x) \geq t}{S \parallel \langle I, L, P, Zc, Zp\rangle \xrightarrow{\text{qry}(I',L',x,v,t)} S' \parallel \langle I, L, P, Zc, Zp \cup \{x\}\rangle} \text{(QRY}_2\text{)}$$

to *confirm* whether the value is up-to-date. Every value in the stigmergy is timestamped, and agents always prefer the value with higher timestamp. Specifically, whenever an agent performs an assignment to a stigmergic variable x, it also records the time when the assignment happened. Then, it adds x to a set of *pending propagation messages* Zp. Similarly, whenever an agent accesses the value of a stigmergic variable y to evaluate an expression, it adds y to a set of *pending confirmation messages* Zc. The operations on Zp, Zc will asynchronously trigger a stigmergic message later. Both propagation and confirmation are constrained by configurable conditions that the sending and the receiving agents must meet. Specifically, two agents may only exchange messages about a variable x if their state satisfies a *link predicate* ψ_x.

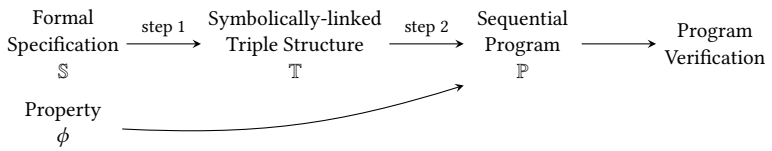
The rules in Table 4 formalize how stigmergic messages are sent and handled. According to rule (PROPAGATE), an agent that has a variable x in its set of pending messages Zp can always remove it from that set by performing a *put*-transition decorated with the value v and timestamp t of x along with the evaluations I and L of the local attributes and stigmergic variables, respectively. Rule (PUT) states how an agent can receive a propagation message for variable x. It has to be running in parallel with a system S which can become S' via a *put*-transition; its state, together with that of the sender, must satisfy the link predicate ψ_x (where x is the variable being propagated from S); furthermore, its own copy L of the stigmergy must contain a timestamp of x lower than the one inside the message. If these conditions are met, the composite system may evolve to a state where S becomes S' and the receiver updates L with the new value of x. Notice that the receiver also adds x to its set of pending messages, so it will contribute to further propagating the value. Confirmation is similar to propagation but uses the timestamp to discard expired data. Rule (CONFIRM) flushes the set of pending messages Zc, exactly as rule (PROPAGATE) does with Zp. Rules (QRY1) and (QRY2) respectively update a stigmergic variable to a newer value or discard it. Notice that in both cases the variable is also added to Zp to propagate the value.

The mechanism described above is not adequate when a user wants to describe structured knowledge (e.g., a pair of coordinates) in the stigmergy with multiple variables, which should be read and updated atomically: since each variable is treated asynchronously and separately, such structured knowledge may be inconsistently communicated to other agents. To avoid this, LAbS

allows to bundle these variables together in a *tuple*, ensuring that they will be treated as a single entity.

3 SEQUENTIAL EMULATION OF DISTRIBUTED SYSTEMS

Before going into the details of our technique, we illustrate the general idea of our methodology. Given the formal specification $\mathbb{S}$ of the system under consideration, we are interested in checking whether a given property ϕ holds. To answer this question, we generate from $\mathbb{S}$ a *sequential* program $\mathbb{P}$. Depending on ϕ, we instrument $\mathbb{P}$ accordingly and perform either reachability or termination analysis on it. By construction, the verification verdict for $\mathbb{P}$ will determine whether ϕ holds for $\mathbb{S}$. This is summarised by the following diagram:



In practice, the input system $\mathbb{S}$ consists of a set of specifications that define the *behaviour* of the different processes, or *agents* [71] of the system, separately. We remark that our *semantics-based* technique is not tied to a particular specification language, but can be applied relatively effortlessly to different languages by taking into account their structural operational semantics (SOS). Intuitively, the SOS rules of the language describe the effect of the actions of an agent on its own state and on the rest of the system, as well as the possible interactions between agents. Throughout this section we assume that the specifications are in CCS, with basic agent interaction via synchronisation. Later on in the paper (Sect. 4), we show that our technique can apply to more sophisticated formal specification languages. Our approach currently supports *invariant properties* and *emergent properties*, i.e., in an LTL-like syntax $\Box\psi$ and $\Diamond\psi$, respectively, where ψ is a predicate over atomic propositions.

Let us first clarify how we formally interpret a temporal property ϕ over a system $\mathbb{S}$. The SOS of the input language associates to $\mathbb{S}$ a labelled transition system $\mathcal{S} \triangleq \langle S, \Lambda, \rightarrow \rangle$. Let AP be a set of atomic propositions (a superset of those that appear in ϕ), and assume that we can always define a subset $I \subseteq S$ of *initial states*, as well as a *successor relation* $R = \{\langle s, s' \rangle \mid \exists \mu . s \xrightarrow{\mu} s'\}$. Then, we can label each state s with a subset $L(s)$ of AP that captures relevant features of that state. In general, this labelling depends on the language in which the system was specified. For instance, if $\mathbb{S}$ is a CCS system, one may simply label each state with the set of actions that may be performed from that state. This would allow to interpret properties related to the reachability of an action μ, such as "$\mathbb{S}$ may never perform a μ-transition". A more complex specification language, e.g., one featuring stateful agents, may require a more elaborate labelling to represent the individual configurations of agents within each state. In any case, once such a labelling has been defined, we obtain a Kripke structure $K \triangleq \langle S, I, R, L \rangle$ representing $\mathbb{S}$: therefore, the question of whether ϕ holds in $\mathbb{S}$ is equivalent to asking whether $K \models \phi$.

To obtain $\mathbb{P}$ we follow a two-step procedure. In the **first step** (Sect. 3.1), we generate a representation $\mathbb{T}$ for $\mathbb{S}$. Such representation $\mathbb{T}$ consists of a set of basic elements, or *transition triples*, corresponding to the elementary sub-expressions of $\mathbb{S}$, i.e., the individual actions of the agents as they occur in their behavioural specifications. Regardless from the actual computation performed by an *action*, at any given time that action may or may not be *enabled*, depending on its position within the formal specifications and on the actions that have been already executed.

For instance, in a sequential composition $a.b$ of two actions, only the first action a is enabled at the very beginning, while the second action b can only enabled after a has been consumed. We can model the only possible execution of this small example by maintaining a *program counter* to keep track of the current execution point, and assigning a unique *identifier* to each of the two actions a and b, for instance 1 and 2, respectively. We initialise the program counter to 1 to enable action a; we set it to 2 right after executing a to enable b, and to 0 right after consuming b, so that a or b cannot be executed again.

We generalise this reasoning to more involved composition operators, i.e., choice and parallel composition, so as to model non-deterministic behaviour and the interleaving of the actions in parallel processes (observe that multiple actions may be enabled at the same time). In general, we can express the possible execution flows allowed by $\mathbb{S}$ in terms of *symbolic expressions* over the values of the program counter right before and right after consuming an action. Such *entry and exit conditions* will capture all the feasible flows of actions in $\mathbb{S}$ by appropriately enabling or disabling the corresponding triples of $\mathbb{T}$. It is important to observe that this representation completely sidesteps an explicit enumeration of the possible transitions between pairs of actions (as it would be in an explicit transition system, for instance), and thus retains the compactness of the original specifications. We refer to $\mathbb{T}$ as a *triple structure*, or *symbolically-linked triple structure* (SLTS) to emphasise this aspect.

In the **second step** of our procedure (Sect. 3.2), we obtain the target program $\mathbb{P}$ by combining $\mathbb{T}$ with the SOS of the input specification language. We add to $\mathbb{P}$ global variables to represent the program counter and some other details about the state of the system. Then, roughly, for each triple in $\mathbb{T}$ we add to $\mathbb{P}$ an *emulation function* that mimics the effect of the corresponding action in $\mathbb{S}$ by manipulating the global variables exactly as prescribed by the SOS rules of the input specification language. Every emulation function is guarded by an *assumption* that enforces the entry condition by appropriately matching the program counter with the unique identifier assigned to the corresponding action in $\mathbb{S}$; similarly, the exit condition is enforced by setting the next value of the program counter right before returning from the emulation function.

We complete $\mathbb{P}$ with its main function, i.e., the *scheduler*, which emulates the feasible executions of $\mathbb{S}$ by repeatedly invoking the emulation functions. The scheduler has the form of an infinite loop. At each iteration, one of the emulation functions is non-deterministically selected and invoked. If the assumption encoding the entry condition at the beginning of the emulation function is satisfied the emulation is carried out, otherwise it is discarded. Note that the scheduler captures the possible interleaving of parallel processes within the behaviour of the same agent, the interleaving of the actions of the different agents, and the interleaving of system-level transitions, if any, with the actions of the agents. Consequently, for each possible execution trace of $\mathbb{S}$ there exists a feasible execution trace of $\mathbb{P}$, and the other way around. Finally, we instrument $\mathbb{P}$ for property checking. In particular, depending on ϕ, we reduce the problem of checking whether $\mathbb{S} \models \phi$ to either reachability in, or termination of, $\mathbb{P}$.

It is worth to remark that our approach is *semantics-based*: to encode a language, one only has to provide the appropriate emulation functions to encode the SOS rules of the input language. If the semantics of the input language specifies additional *system-level transitions*, e.g., for synchronisation events that may be triggered in between two actions in $\mathbb{S}$, we introduce additional emulation functions for them; precisely, we introduce one emulation function for each SOS rule. The exit condition for such emulation function will not constrain the program counter (as system-level transitions such as communication events can be triggered asynchronously, and in any case depending on the semantics), but will simply encode the very same condition expressed by the premise of the specific SOS rule being modelled.

3.1 From Formal Specifications to Symbolically Linked Triples

In this section, we define the concepts of triple and triple structure and, based on them, we give a formal definition of our transformation from $\mathbb{S}$ to $\mathbb{T}$. Intuitively, each triple of $\mathbb{T}$ symbolically represents (possibly multiple) transitions in the LTS of the encoded process. The execution of $\mathbb{S}$ is modelled by keeping track of which transitions may be performed at any given time. This is achieved by equipping the triple structure with a *program counter*, by guarding each triple with an appropriate predicate over it, and by updating the program counter right after executing a transition, so as to correctly enable the next (possibly multiple) feasible transitions.

Definition 3.1 (Transition triple). Let the *program counter* $pc = \langle pc_0, pc_1, \ldots, pc_l \rangle$ be a vector of integer variables. For the time being, let us assume that the vector is arbitrarily long (i.e., that l is arbitrary). Later on, we will see how its length can be constrained. A *transition triple* (or *triple*, for short) t consists of an *entry condition*, an *action*, and an *exit condition*, and is denoted as follows:

$$t = \langle \triangleright(t), \mu(t), \triangleleft(t) \rangle.$$

The *entry condition* $\triangleright(t) = (\triangleright_0(t) \wedge \triangleright_1(t) \wedge \cdots \wedge \triangleright_l(t))$ is a predicate over pc that specifies the required condition at the beginning of the transition, i.e., t is *enabled* iff. $pc \models \triangleright(t)$. Each conjunct $\triangleright_i(t)$ can only predicate on the possible values of pc_i or remain *unconstrained*, i.e., $\triangleright_i(t) = (\cdot)$. The *action* $\mu(t)$ represents either an elementary action of $\mathbb{S}$ or a *null action* λ. The *exit condition* $\triangleleft(t) = (\triangleleft_0(t) \wedge \triangleleft_1(t) \wedge \cdots \wedge \triangleleft_l(t))$ is a predicate that defines the condition of pc at the end of the transition. We write $\triangleleft_i(t) = (\cdot)$ to denote that an exit condition $\triangleleft(t)$ leaves the element pc_i unconstrained.

Definition 3.2 (Symbolically-linked triple structure). A *symbolically-linked triple structure (triple structure)* is a pair $\mathbb{T} = \langle T, pc \rangle$, where T is a set of transition triples and pc is a program counter. The *evolution condition* of the triple structure is the least relation induced by the following inference rule:

$$\frac{t \in T \quad pc \models \triangleright(t) \quad pc' \models \triangleleft(t) \quad \triangleleft_i(t) = (\cdot) \Rightarrow pc'_i = pc_i}{\langle T, pc \rangle \xrightarrow{\mu(t)} \langle T, pc' \rangle}$$

At any step, zero or more triples can be enabled depending on the current value of pc and on their entry conditions. Any enabled triple of $\mathbb{T}$ may (or may not) *evolve* by performing the action $\mu(t)$ and thus becoming $\langle T, pc' \rangle$.

The first two premises control the set of enabled triples in $\mathbb{T}$, i.e., those triples whose entry condition is satisfied by the program counter pc. The last two premises define the set of enabled triples after consuming $\mu(t)$.

Observe that the last premise combines the actual value of pc with the exit condition by replacing each unconstrained conjunct in $\triangleleft(t)$ with an equality check on the initial value of the corresponding element of pc. Therefore, the execution of two triples with the same exit condition will not necessarily yield the same sets of enabled triples, as it does depend on the value of pc as well. For any triple t' enabled in $\langle T, pc' \rangle$, we say that t is *symbolically linked* to t'.

We now introduce an encoding function $[\![\cdot]\!]$ that maps process terms to triple structures. Let us assume that each elementary sub-expression μ of the encoded process term is given a unique *identifier*, i.e., a positive number $id(\mu)$. Notice that, even though two sub-expressions may be identical, their identifiers are still distinct. For instance, if the encoded process contain several occurrences of the same action, we will reserve a separate identifier for each occurrence. The encoding function considers three process composition operators: binary sequential composition $(P; Q)$; n-ary nondeterministic choice $(\Sigma_i P_i)$; and n-ary parallel composition $(\Pi_i P_i)$. We treat action

prefixing (denoted $a.P$ in CCS) as a special case of sequential composition. Processes within a parallel composition may perform two-party synchronisation according to a (language-specific) synchronisation algebra. The encoded process may also contain *process constants*. To simplify our exposition, we assume that these constants always refer to the process itself, but this procedure can be easily generalised by recursively calling it to encode also constants that refer to other processes. We also restrict the use of these constants by disallowing recursion within parallel composition (as in $P \triangleq a.0 \mid b.P$), as well as unguarded recursion (as in $P \triangleq P + a$).

Elementary examples. Before defining the encoding formally, let us consider a couple of elementary processes (Figs. 1(a) and 1(b)). In the first process, an action is followed by a non-deterministic choice between two further actions. In the second process, three actions are executed in parallel. The two processes can be expressed in CCS respectively as $a.(b.0 + c.0)$ and $a.0 \mid b.0 \mid c.0$. The figure shows the LTSs of both processes, and the corresponding triple structures generated by our encoding. Each triple is represented by a box containing the entry condition on the left, the action in the middle, and the exit condition on the right. An edge between two triples denotes that they are symbolically linked.

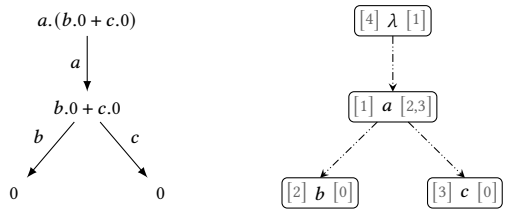
Please notice that in Fig. 1(a), we assign to each triple a unique identifier and, to model the fact that after action a (with id 1) the process can perform either b or c (with ids 2 and 3, respectively), we put the ids of the triples corresponding to b and c on the right of the a-triple.

In general, we represent entry and exit conditions of a triple as vectors of the same size as the number of parallel processes plus one. In fact, the different elements of the vectors predicate over the different elements of pc. Specifically, an integer k at the i-th position of the vector stands for the predicate $pc_i = k$, while the $\cdot$ symbol means that pc_i is unconstrained. A triple is enabled if all its predicates hold. For example, in the SLTS of Fig. 1(a) the program counter has only one element, pc_0, and the a-triple is only enabled when pc_0 is 1. The SLTS of Fig. 1(b) has a program counter of four elements. The c-triple requires pc_3 to be 1 in order to be enabled, regardless from the values of pc at the other positions. The exit conditions are similar to the entry conditions, but in addition may contain multiple values within the same element. For example, after executing the a-triple of Fig. 1(a), pc_0 is non-deterministically assigned either 2 or 3. Note that many triples may be enabled at the same time, and that each triple may be symbolically linked to many triples; conversely, many triples may be linked to the same triple.

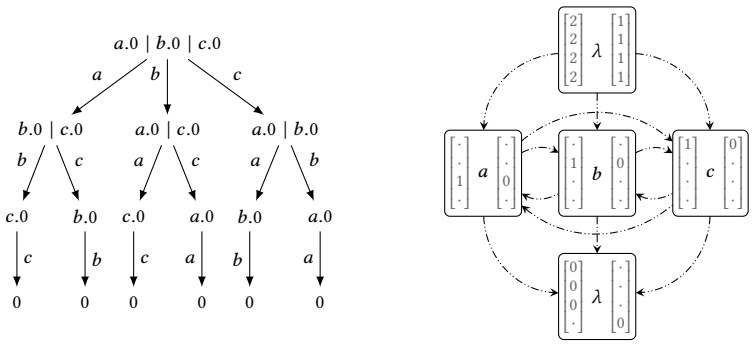
Notice at the top of both the SLTSs of Figs. 1(a)–1(b) the triples for a null action λ. We call them *start triples*. A start triple does not correspond to any concrete action in $\mathbb{S}$, and is guaranteed to be executed exactly once, at the very beginning of the emulation. To ensure that, we initialise the program counter to the (unique) identifier of that triple. In the SLTS of Fig. 1(a) the start triple is enabled when pc_0 is 4. The exit condition of the start triple is $pc_0 = 1$, which enables the *initial* action a of the encoded process. Throughout the section we will occasionally denote with $t^\star$ the start triple, and with $pc^\star$ the value of the program counter that enables it. For process termination, we set pc to 0, as in the b-triple and the c-triple of Fig. 1(a), and in the λ-triple at the bottom of Fig. 1(b).

The process of Fig. 1(a) must necessarily perform a as its first action. Therefore, we symbolically link the start triple to the a-triple by appropriately matching the exit condition of the λ-triple with the exit condition of the a-triple. In turn, the exit condition of the a-triple may enable either the b-triple or the c-triple, to model the non-deterministic choice operator +. The other two triples instead update pc_0 to 0 to denote that the process terminates after performing either of them.

Fig. 1(b) shows how a parallel process is encoded by a triple structure of four triples and a program counter of length 4. The first component of the program counter, pc_0, tracks the execution of the overall process Π (the *parent*), while each of the other component $pc_{1,2,3}$ tracks one of the



(a) $a.(b.0 + c.0)$.



(b) $a.0 \mid b.0 \mid c.0$.

Fig. 1. LTSs and triple structures for two simple CCS processes.

sub-processes within the parallel composition (the *children* of Π). Process Π may perform any permutation of actions a, b, and c. Therefore, the exit condition of the start triple enables all their corresponding triples. We also track the termination of Π by adding a *join triple* t_{join} that has entry condition $pc_1 = pc_2 = pc_3 = 0$ and no action. This entry condition is only satisfied when all children of Π have terminated. If this is the case, the triple sets the element pc_0 to 0 so as to signal that Π itself has terminated as well. This addition may not seem necessary at this point, but is needed when the parallel process is sequentially composed with another process $(\Pi; Q)$. In this case, the join triple updates the program counter so that the Q process may start upon termination of Π.

Notice that each μ-triple ($\mu = a, b, c$) symbolically represents all μ-transitions of the LTS of Π. The vectorial program counter keeps track of all interleavings within the parallel composition without explicitly representing them. This retains compactness.

Encoding sequentiality and choices. Let us start by considering processes with no parallel composition. To encode these processes, a program counter with a single component pc_0 is sufficient. An action μ is encoded as a triple that has entry condition $pc_0 = id(\mu)$ and action μ. The exit condition of this μ-triple depends on the actions that may follow μ in the encoded process. If μ is followed by a generic process term P, we can define a function $\lhd [P]_0$ returning an exit condition over pc_0. This exit condition only enables the correct triples generated from P, i.e., those corresponding to actions that may directly follow μ. Thus, we call $\lhd [P]_0$ the *enabler* of P. To model recursion, we simply compute the enabler of the entire process and use it as the exit condition of the triples preceding the recursive call. For instance, consider a process $P \triangleq a.P$. The exit condition of the a-triple is $\lhd [P]_0 = \lhd [a.P]_0$. In general, we define the enabler of a sequential composition $P; Q$ to be the enabler of P: intuitively, this means that the triples generated from Q cannot be enabled until P has

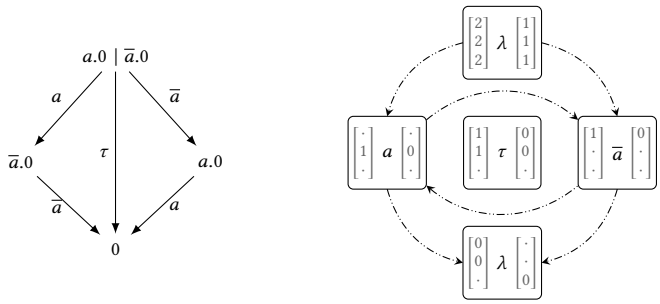


Fig. 2. LTS and triple structure for a process with synchronisation.

terminated. Since we treat action prefixing as a specific case of sequential composition, we apply this definition to obtain $\triangleleft\,[a.P]_0 = \triangleleft\,[a]_0$, which is the entry condition of the a-triple. Therefore, the a-triple enables itself after being performed, and the recursive nature of P is encoded properly.

To illustrate the procedure in detail, let us see how the process $a.(b.0 + c.0)$ is encoded as the triple structure shown in Fig. 1(a). Let us assume that $id(a) = 1$, $id(b) = 2$, $id(c) = 3$, and let us start by encoding the $b.0$ sub-term. To do so, we encode b and 0 separately and then use the enabler of 0 as the exit condition of b. The encoding $[\![b]\!]$ is a single triple with entry condition $\triangleright[b]_0 \triangleq pc_0 = 2$; on the other hand, 0 is the deadlocked process and thus its encoding is the empty set. We define its enabler to be $\triangleleft\,[0]_0 \triangleq pc_0 = 0$, which we use as the exit condition of the b-triple. We encode the sub-term $c.0$ likewise. Finally, we encode the sequential composition of a with the choice $b.0 + c.0$. First, we construct an a-triple with entry condition $pc_0 = 1$. Then, we compute the enabler of the choice term as the disjunction of the enablers of its sub-terms ($pc_0 = 2 \vee pc_0 = 3$). Informally, this exit condition will set pc_0 to either 2 or 3, enabling either the b- or the c-triple and thus mimicking the choice operator.

Encoding of a parallel process. Let us now consider processes that contain a parallel composition operator, such as the one shown in Fig. 1(b). To encode such a process, we give additional, unique identifiers $id(\Pi_i P_i)$, $id(P_i)$ to all parallel composition terms and their sub-terms. Each child process within a parallel composition $\Pi \triangleq \Pi_i\ P_i$ may evolve independently of the others. To model that, we encode each child process as a separate set of triples, whose entry and exit conditions consider separate elements $pc_{id(P_i)}$ of the program counter. The enabler of Π is the conjunction of the enablers of its children: this encodes the fact that all children are enabled from the start. Furthermore, we add to the sequence another clause $pc_k = id(\Pi)$. Intuitively, by setting pc_k to such a value, we guarantee that only triples from within Π may be performed until Π terminates. Notice that, until now, we assumed pc to be arbitrarily long. Now, we can constrain its length l by the maximum process identifier assigned to Π or one of its sub-terms.

This encoding captures the *interleaving* of the children processes, which progress by alternating the execution of their actions. However, many process algebras also allow parallel processes to *synchronise* on pairs of actions. Let us assume that there is a synchronisation algebra σ that formalises the synchronisation semantics of the input language. Our approach to model two-party synchronisation, then, is to first compute the interleaving triple structure, and then insert a transition for each pair of actions that may synchronise. We illustrate this approach on the simple CCS process $a.0 \mid \overline{a}.0$ (Fig. 2). Its triple structure is obtained by first computing the interleaving triple structure, resulting in one a-triple t_a and one $\overline{a}$-triple $t_{\overline{a}}$. Then, we introduce a third triple with entry condition $\triangleright(t_a) \wedge \triangleright(t_{\overline{a}})$, action $\sigma(a, \overline{a}) = \tau$, and exit condition $\triangleleft(t_a) \wedge \triangleleft(t_{\overline{a}})$. By definition, whenever t_a and $t_{\overline{a}}$ are enabled, so is this newly created τ-triple. If the τ-triple is performed, its

Table 5. Definition of the translation function $[\![\cdot]\!]$.

$$
\begin{aligned}
[\![0]\!]_k^{\triangleleft} &\triangleq \emptyset \\
[\![K]\!]_k^{\triangleleft} &\triangleq \emptyset \\
[\![\mu]\!]_k^{\triangleleft} &\triangleq \left\{\left\langle pc_k = id(\mu),\ \mu,\ \triangleleft\right\rangle\right\} \\
[\![P;Q]\!]_k^{\triangleleft} &\triangleq [\![P]\!]_k^{\triangleleft[Q]_k} \cup [\![Q]\!]_k^{\triangleleft} \\
[\![\Sigma_i P_i]\!]_k^{\triangleleft} &\triangleq \bigcup_i [\![P_i]\!]_k^{\triangleleft} \\
[\![\Pi_i P_i]\!]_k^{\triangleleft} &\triangleq \bigcup_i T_i \cup \bigcup_{i\neq j} \left\{ [\![t_i, t_j]\!]_\sigma \,\middle|\, t_i \in T_i, t_j \in T_j, \sigma(\mu(t_i), \mu(t_j)) \neq \bot \right\} \cup t_{join} \\
\text{where} \quad T_i &= [\![P_i]\!]_{id(P_i)}^{(pc_{id(P_i)}=0)} \\
[\![t, t']\!]_\sigma &= \langle \triangleright(t) \wedge \triangleright(t'), \sigma(\mu(t), \mu(t')), (\triangleleft(t) \wedge \triangleleft(t'))\rangle \\
t_{join} &= \left\langle \bigwedge_i \left(pc_{id(P_i)} = 0\right), \lambda, \triangleleft \right\rangle
\end{aligned}
$$

exit condition updates the current program counter according to both $\triangleleft(t_a)$ and $\triangleleft(t_{\overline{a}})$, therefore setting both pc_1 and pc_2 to 0. Then, the join triple becomes enabled and may set pc_0 to 0 as well, denoting that the parallel process has terminated.

Formal definitions. We formally define our *translation* from a behaviour S to a set of triples via a function $[\![P]\!]_k^{\triangleleft}$ (Table 5), where P is a syntactic fragment of S, k is a program counter index, and $\triangleleft$ is an exit condition. Both the deadlocked process 0 and the recursive invocation K translate to the empty set. The translation $[\![\mu]\!]_k^{\triangleleft}$ of a single action μ only contains one triple. This triple has an entry condition checking that pc_k matches the identifier of μ, action μ, and exit condition $\triangleleft$. To translate a sequential composition $P;Q$, we translate the two processes separately and use the enabler of Q as the exit condition parameter when translating P. The translation of a choice is the union of the translations of its terms. Finally, translating the parallel composition of n process terms P_i requires translating each P_i with a different program counter index, namely $id(P_i)$. Then, a synchronisation triple is added for each pair of triples corresponding to synchronising actions. Lastly, the join triple must be added, so that, when all child processes have terminated, the exit condition $\triangleleft$ of the parent may be applied. Finally, Table 6 contains a formal definition of the enabler function which has been informally described thus far. Both definitions are given by induction on the structure of syntactic fragments P.

Definition 3.3 (Process encoding). Given a process S, let $T = [\![S]\!]_0^{pc_0=0}$ and let $pc^\star$ be a program counter that does not enable any triple in T. Then, the *encoding* of S is the triple structure $\mathbb{T} = \left\langle T \cup \{t^\star\}, pc^\star \right\rangle$, where $t^\star \triangleq \left\langle pc = pc^\star, \lambda, \triangleleft\, [S]_0 \right\rangle$.

System-level encoding. We consider a *system* $\mathbb{S}$ to be a composition of concurrent processes (i.e., the agents), which we assume to be different from each other and possibly recursive. We also assume that agents may synchronise on the pairs of actions specified in a synchronisation algebra $\sigma_{\mathbb{S}}$.

Definition 3.4 (System encoding). Let $\mathbb{S} = \{S_1, \ldots, S_n\}$. Let $\mathbb{T}_i = \langle T_i, pc^{i\star}\rangle$ the encoding of S_i. The *encoding* of $\mathbb{S}$ is a triple structure

$$\langle T \cup T_\sigma, pc^\star \rangle$$

Table 6. Definition of the enabler function $\lhd [\cdot]_k$.

$$
\begin{aligned}
\lhd [0]_k &\triangleq \big(pc_k = 0\big) \\
\lhd [K]_k &\triangleq \lhd [P]_k \text{ iff. } K \triangleq P \\
\lhd [\mu]_k &\triangleq \big(pc_k = id(\mu)\big) \\
\lhd [P;Q]_k &\triangleq \lhd [P]_k \\
\lhd [\Sigma_i P_i]_k &\triangleq \bigvee_i \lhd [P_i]_k \\
\lhd [\Pi_i P_i]_k &\triangleq (pc_k = id(\Pi_i P_i)) \wedge \bigwedge_i \lhd [P_i]_{id(P_i)}
\end{aligned}
$$

where T is the union of all T_i, with appropriate adjustments to the entry and exit conditions of triples to avoid spurious symbolic links; T_σ contains a triple for each pair of triples $t_i \in T_i, t_j \in T_j$ such that $i \neq j$ and the actions of the two triples may synchronize according to $\sigma_{\mathbb{S}}$; $pc^\star$ is the concatenation of all $pc^{i\star}$.

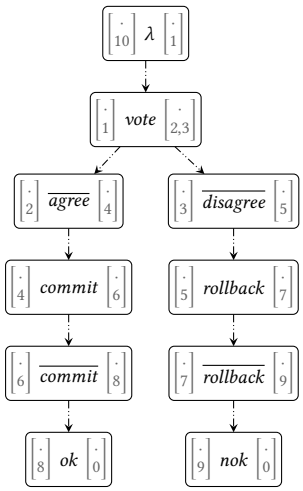
Intuitively, when we construct T we need to adjust entry and exit conditions of each T_i so that they refer to the correct components of the concatenated program counter.

To illustrate the system encoding procedure, we now describe the triple structure (shown in Fig. 3) of the $2PC_{CCS}$ system described in Example 2.1. It includes the individual triple structure of each agent, namely Coord (Fig. 3(a)) and Worker (Fig. 3(b)), and an additional set of τ-triples, one for each pair of complementary actions (Fig. 3(c)). The program counter of the whole system has two components which track the evolution of Worker and Coord, respectively. The $2PC_{CCS}$ system admits several traces leading to *nok*. These traces are still present in its triple structure: one of them, namely $\langle \tau, \overline{disagree}, agree, \tau, \overline{rollback}, nok \rangle$, is graphically represented in Fig. 4. Each diagram in the figure represents the triple structure of Fig. 3: black circles denote enabled triples.

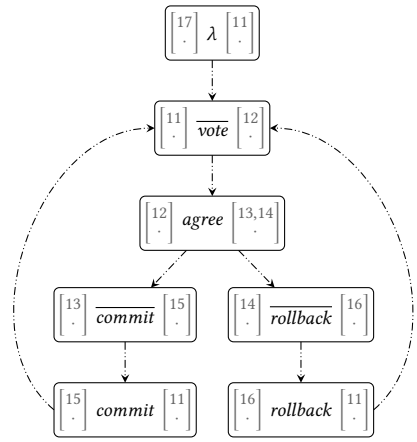
Language-specific interaction rules. In some process algebras there is a strict difference between terms that may freely occur within the structure of a process, and terms that may only appear at the top level. These operators typically describe additional rules related to the interaction between agents. For instance, we may see the CCS restriction operator as one such operator. This does not lead to any loss of generality: a CCS process with any number of restriction operators may always be expressed as a process with a single restriction operator, by means of suitable α-renamings to avoid unwanted name capture.

Encoding a restricted system $\mathbb{S} \setminus \Theta$ is straightforward: we simply encode $\mathbb{S}$, then remove any triple whose action is (equal or complementary to) a member of Θ. To illustrate this procedure, let us now consider the variant $2PC'_{CCS}$ of our two-phase commit scenario. The triple structure of the restricted 2PC system is obtained from the one of Fig. 3 by removing all triples except the ones encoding *ok*, *nok*, and the synchronisations. This makes the *nok* triple unreachable from the start triple.

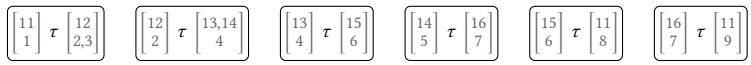
Supporting multi-party synchronisation. The encoding function of Table 5 assumes that agents are not necessarily required to synchronise with each other in order to perform an action, and that synchronisation involves at most two processes. This is enough for CCS, but cannot encode more complex mechanisms, such as Hoare's operator. Algorithm 1 sketches a possible approach to generate the set of triples for a process P defined as n processes composed together with Hoare's operator: $P \triangleq P_1 \,|[L]|\, \ldots \,|[L]|\, P_n$. In the rest of this paragraph we will also use the shorter notation $P \triangleq \big|[L]\big|_{i=1}^{n} P_i$.



(a) Coord.



(b) Worker.



(c) Synchronisation triples.

Fig. 3. Triple structure generated from the specifications of the two-phase commit (2PC) example.

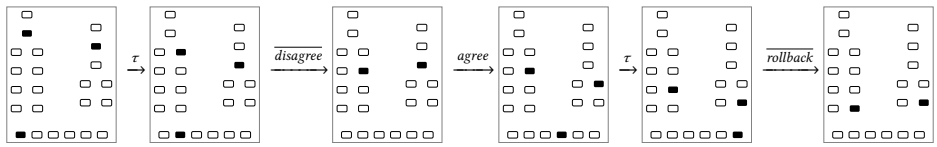


Fig. 4. Graphical representation of a possible execution of the 2PC example.

First, a set of triples T_i is generated for each process P_i (line 1). Then, we accumulate a set T of triples that initially contains all triples in T_1 whose action may synchronise with some other action α (line 2). Then, for each T_i, $i = 2, \ldots, n$, we accumulate two set of triples which will be either inserted into T (I) or removed from it (R). Both are initially empty (line–4). For each pair of triples (t, t') in $(T \times T_i)$ such that their actions may synchronise, we add a synchronisation triple to I (line 6) and add the triple t to R (line 7). We also add to R all triples $t \in T$ such that $\mu(t)$ cannot synchronise with any triple in T_i (lines 9–12). Intuitively, we need to remove such triples because Hoare's parallel operator requires *all* processes to synchronise on the same action. If T_i contains no triple with such an action, it means that P_i is not willing to perform that action. Therefore, a triple $t \in T$ that cannot synchronise with any triple in T_i encodes a transition that cannot happen, and thus must be removed. At the end of each iteration of the outer loop, we update T by removing all triples that also belong to R, and then adding the triples from I (line 14). Once all T_i have been processed, we also add to T those triples of $T_1, \ldots, T_n$ whose actions are not forced to synchronise (lines 16–18), and finally add the join triple (line 19).

Having described how Algorithm 1 works, we now state two lemmas concerning its time complexity and its correctness when computing the SLTS for a process $P \triangleq \left|\left[L\right]\right|_{i=1}^{n} P_i$. Intuitively,

Algorithm 1. Encoding Hoare's parallel operator as a set of transition triples.

input : A process $P \triangleq |[L]|_{i=1}^{n} P_i$
A synchronisation algebra σ that captures $\cdot\,|[L]|\,\cdot$
output: A set of transition triples T

1 $T_i \leftarrow [\![P_i]\!]_{id(P_i)}^{(pc_{id(P_i)}=0)}$, $i = 1, \ldots, n$
2 $T \leftarrow \{t \in T_1 \mid \exists \alpha \neq *.\sigma(\mu(t), \alpha) \neq \bot\}$
3 **foreach** $i = 2, \ldots, n$ **do**
4 $\quad I \leftarrow \emptyset, R \leftarrow \emptyset$
5 $\quad$ **foreach** $(t, t') \in T \times T_i.\sigma(\mu(t), \mu(t')) \neq \bot$ **do**
6 $\quad\quad I \leftarrow I \cup [\![t, t']\!]_\sigma$
7 $\quad\quad R \leftarrow R \cup t$
8 $\quad$ **end**
9 $\quad$ **foreach** $t \in T$ **do**
10 $\quad\quad$ **if** $\forall \alpha.\sigma(\mu(t), \alpha) \neq \bot.\forall t' \in T_i.\mu(t') \neq \alpha$ **then**
11 $\quad\quad\quad R \leftarrow R \cup t$
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 $\quad T \leftarrow (T \setminus R) \cup I$
15 **end**
16 **foreach** $t \in \{T_1 \cup \cdots \cup T_n\}.\sigma(\mu(t), *) \neq \bot$ **do**
17 $\quad T \leftarrow T \cup t$
18 **end**
19 $T \leftarrow T \cup t_{join}$

Lemma 3.5 states that Algorithm 1 runs in polynomial time, while Lemma 3.6 shows that each triple within the output of the algorithm represents either an action that does not belong to L, or one that is in L and upon which all processes are willing to synchronise. Thus, T correctly emulates the behaviour of Hoare's parallel operator.

Lemma 3.5 (Time complexity of Algorithm 1). *Let $P \triangleq |[L]|_{i=1}^{n} P_i$, and let σ be a synchronisation algebra that captures $\cdot\,|[L]|\,\cdot$. Assume that each P_i contains at most m actions, and that no two actions within each P_i may synchronise. Then, Algorithm 1 with input $\langle P, \sigma \rangle$ terminates in time $O(n \cdot m^2)$.*

Proof. The lack of synchronisation within each P_i implies that each triple structure T_i contains no synchronisation triples. Thus, each T_i contains at most $O(m)$ triples and may be generated in time $O(m)$. Therefore, both line 1 and the loop at lines 16–18 run in time $O(n \cdot m)$. Let us now focus on the loop at lines 3–15. The inner loop at lines 5–8 features a Cartesian product between two triple structures T, T_i. As said above, T_i contains $O(m)$ triples. We claim that T also contains $O(m)$, for all iterations of the outer loop. In fact, T is initialised to a subset of T_1, so it contains $O(m)$ triples before the first iteration of the outer loop (line 2). At each iteration, T is updated by adding and removing triples that belong to sets I and R, respectively (line 14). Notice, however, that I and R are empty at the beginning of each iteration of the outer loop (line 4), and that every time time a triple is added to I one is also added to R (lines 6–7). Therefore, T always contains $O(m)$ elements. The inner loop, then, must check $O(m) \cdot O(m)$ pairs of triples, and thus runs in time $O(m^2)$. Therefore, the outer loop runs in time $O(n \cdot m^2)$ and dominates the overall running time of the algorithm. □

Lemma 3.6 (Correctness of Algorithm 1). *Let $P \triangleq \left|\left[L\right]\right|_{i=1}^{n} P_i$, σ a synchronisation algebra that captures $\cdot\,|[L]|\,\cdot$, and T the triple structure obtained by running Algorithm 1 on input $\langle P, \sigma\rangle$. Additionally, let $T_1, \ldots, T_n$ the triple structures generated after executing line 1 of Algorithm 1. Then, a triple t belongs to T if and only if one of the following holds:*

(1) *t belongs to some T_i and $\sigma(\mu(t), *) \neq \bot$;*
(2) *t is the join triple;*
(3) *there exist $t_1 \in T_1$, $t_2 \in T_2$, …, $t_n \in T_n$ such that $t = [\![\ldots [\![[\![t_1, t_2]\!]_\sigma, \ldots]\!]_\sigma, t_n]\!]_\sigma$.*

Proof. All triples that satisfy (1) or (2) are added to T at lines 16–19. To show that (3) is a sufficient condition for t being in T, let us consider how triples are added to T throughout the algorithm. Initially, T contains those triples of T_1 whose actions may synchronise according to L (line 2). Whenever we find within some T_i a triple t' that may synchronise with some triple $t \in T$, we replace t with the synchronisation triple $[\![t, t']\!]_\sigma$ (lines 5–8). Furthermore, if for some $t \in T$, some T_i contains no such triple, we remove t from T (lines 9–12). Therefore, when the loop at lines 3–15 ends, only those triples that satisfy (2) are in T.

Since no other triples are added to T, a triple that satisfies neither (1), nor (2), nor (3) cannot be in T when the algorithm terminates. □

3.2 From Symbolically Linked Triples to Imperative Programs

We now describe how to generate the *emulation program* $\mathbb{P}$ for the system $\mathbb{S}$, relying on $\mathbb{T}$ as an intermediate representation for the individual behaviours of the agents. The structure of $\mathbb{P}$ is shown in Listing 1, where `N` denotes the total number of agents in $\mathbb{S}$, `pc[N]` the program counter, and `agent`, `action` respectively the identifier of the agent and of the action of $\mathbb{S}$ currently being emulated.

Separate *emulation functions* in $\mathbb{P}$ encode the transition triples obtained by the procedure from Section 3.1. The program in the figure represents a simplified version of the encoding for 2PC system described in the previous section and whose SLTS is shown in Fig. 3. For instance, function `vote` emulates the triple *vote* in Fig. 3(a). The `assume` statement at the beginning implements the entry condition of the triple (see e.g. line 4). The nondeterministic assignment and the `assume` statement at the end implement the exit condition. For conciseness we do not report the other emulation functions. When dealing with more sophisticated SOS rules, it may be necessary to define additional global variables to represent other information about the state of $\mathbb{S}$. The emulation functions can manipulate such global variables following the operational semantics of the specification language. We will further discuss the details of implementing emulation functions in Section 4.

If multiple agents with the same behaviour are defined in $\mathbb{S}$, we only need to encode that behaviour into emulations functions once. We additionally introduce a global variable `id` that the emulation functions can access to determine which agent is performing the action (not shown in the program). Thus, we can concisely represent systems that contain multiple agents with the same behaviour. Although not shown in Listing 1, one could use dynamically allocated arrays to store the program counters of the agents and other information about their state. This type of encoding would enable us to model agents that are able to *spawn* new agents as part of their behaviour, giving rise to systems of dynamic (possibly unbounded) size.

In the rest of the encoding of Listing 1, we embed the scheduler (lines 28–41) along with a few ancillary functions. The scheduler initialises the global state of the program by invoking function `init()` (line 29), that also initialises the program counters of the agents according to the exit conditions of their start triples. This is equivalent to implementing each start triple as an emulation function and initialising the program counters according to $pc^\star$.

At each iteration of the main loop, the scheduler nondeterministically picks an agent to simulate, say `agent` = i, by invoking `next()` (lines 22–26). Then, it simulates an action of S_i by invoking the

Listing 1. Emulation program for the 2PC system.

```
1  int pc[N], agent, action;
2
3  vote() {
4       assume(pc[agent] = 1);
5       action := vote;
6       pc[agent] := *;
7       assume(pc[agent] = 2 ∨ pc[agent] = 3);
8  }
9
10 ...        // Other emulation functions
11
12 init() {
13      pc[1] := 11;  // Worker
14      pc[0] := 1;  // Coord
15 }
16
17 check() {
18      if(¬always) error;
19      if(eventually) exit;
20 }
21
22 next() {
23      if (fair) agent := (agent + 1) % N;
24      else agent := *;
25      assume(agent < N);
26 }
27
28 main() {
29      init();
30
31      while (true) {
32           next();
33
34           choice := *;
35           if (choice = 1) vote();
36           if (choice = 2) agree();
37           ...
38
39           check();
40      }
41 }
```

corresponding emulation function. Any of the emulation functions can be selected at each iteration of the scheduler. However, the assumptions on the program counter of `agent` will prune away unfeasible executions by enforcing the entry conditions.

The nondeterministic choice of the agent to simulate models the interleaving of the behavioural processes of the agents. In addition, the scheduler may nondeterministically attempt to invoke a system-level emulation function (e.g., synchronisation in CCS). This linearises the concurrent

execution of agent-level transitions $(S_1, \ldots, S_n)$ and of system-level transitions and yields different advantages. First, it models the interleaving compactly: thanks to symbolic expressions and nondeterministic updates to the program counters, it can represent an exponential number of feasible executions. Second, it removes concurrency and therefore allows to use program analysis techniques that only support sequential programs. Third, it allows to model scheduling variations by simply restricting the nondeterminism over the interleaving of agents, i.e., by overriding the `next()` function. Currently, we provide a completely nondeterministic scheduler as well as a round-robin one, depending on the flag `fair`.

A function `check()` encodes the property ϕ to verify (lines 17–20). The code sample shows both types of encodings; however, in practice only one property at a time is encoded. In the case of an invariant property, we simply add the formula to the program as an assertion, which is checked at every step of the scheduler. For instance, if we want to instrument the program of Listing 1 to verify whether the *fail* action is unreachable in the 2PC system, we simply replace the body of function `check()` with the statement `if (action = fail) error;`.

To deal with emergent properties, we use the state formula as a termination condition for the whole program. Then, we can perform termination analysis on the generated program. Since the program can only terminate when ϕ holds, verifying that the program unconditionally terminates is equivalent to verifying that ϕ holds in $\mathbb{S}$. In our example program, we can check whether an *ok* action is always reached in the 2PC system with the statement `if (action = ok) exit;`.

If the source language defines agents with individual states, we can easily support properties over them, possibly containing existential and universal quantifiers over the agents. To do so, we first apply quantifier elimination and then encode the resulting first-order formula into a predicate over the variables of $\mathbb{P}$ that encode the state of the agents.

Size of the emulation program. The encoding function of Sect. 3.1 generates at most one triple for each elementary action in the system (triples corresponding to restricted actions are removed), plus one triple for each pair of synchronising actions. The emulation program, then, will contain one emulation function for each of these triples. The size of the program counter for each agent is linear in the number of parallel processes in its behaviour.

Translation of Counterexamples. We now discuss counterexample generation for safety property violations. By construction, if a given safety property of interest can be violated in the system $\mathbb{S}$ under analysis, the emulation program $\mathbb{P}$ will contain a reachable assertion failure (we will sketch a proof of this claim in Section 3.3). In that case, it may be possible to obtain a counterexample trace for $\mathbb{S}$, depending on the specific technique and tool employed for the analysis of $\mathbb{P}$. In general, if a counterexample for $\mathbb{P}$ is precise enough, it can be translated into one that refers to $\mathbb{S}$.

Clearly, the program state in the untranslated counterexample will mix the variables of the emulation program that represent important features of the original system (e.g, the agents' attributes, or their environment) with the variables used by the scheduler (`pc`, `action`, `choice`, ...) as well as other variables (if any) introduced in the emulation functions to implement the semantics of the source language (such as the global variables declared at the beginning of Listing 2). Intuitively, the last group of variables (and transitions involving them) are irrelevant to the end user and thus should be discarded[2], while the transitions on the schedulers' variables allow to reconstruct the sequence of active processes performing an action in $\mathbb{S}$, and thus provide the skeleton of the counterexample for $\mathbb{S}$. The counterexample is completed by decorating this skeleton with the state

[2]We are assuming that the operational semantics and their emulation functions are correct. If one is interested in debugging the operational semantics rules, then it would be useful not to remove those transitions.

transitions resulting from applying the individual actions as described in the system specifications and prescribed by the operational semantics.

More concretely, let us assume that the counterexample $\pi_{\mathbb{P}}$ for the emulation program is a sequence of assignments represented as pairs $(var_{\mathbb{P}}, v)$, where $var_{\mathbb{P}}$ is the name of some variable in the emulation program and v its assigned value. In this case, what we want to obtain is a translated trace $\pi_{\mathbb{S}}$ consisting of triples $(id, var_{\mathbb{S}}, v)$, denoting that the agent with identifier id is assigning v to $var_{\mathbb{S}}$. To that end, during the generation of $\mathbb{P}$, we build a lookup table l that maps each variable of $\mathbb{P}$ to either a variable of $\mathbb{S}$ or to $\bot$, if the variable is irrelevant. Keeping in mind that $\mathbb{P}$ uses a variable `agent` to store the id of the agent currently selected by the scheduler, we translate $\pi_{\mathbb{P}}$ into $\pi_{\mathbb{S}}$ according to the following procedure:

(1) discard all pairs $(var, v) \in \pi_{\mathbb{P}}$ such that $var \neq$ `agent` and $l(var) = \bot$;
(2) for every remaining pair (var, v):
  (a) if $var =$ `agent`, store v in a variable id;
  (b) otherwise, append $(id, l(var), v)$ to $\pi_{\mathbb{S}}$.

3.3 Correctness Sketch

We now sketch a correctness proof for our two-step encoding. Intuitively, such proof shows that, given a system $\mathbb{S}$ and an invariant property ϕ, an error state is reachable in the emulation program $\mathbb{P}$ generated from them if and only if $\mathbb{S}$ violates ϕ.

As shown in Listing 1, $\mathbb{P}$ is composed of a scheduler, and separate functions to emulate the transitions of $\mathbb{S}$. The user has full responsibility of the actual implementation of the emulation function corresponding to each action. When reasoning about correctness of our encoding, we simply assume correctness of her/his implementation. At each iteration, the scheduler emulates a transition by calling one of these functions. The interleaving between agents is modelled by the nondeterministic assignment to the `agent` variable (lines 22–26). State variables are only altered within an emulation function, and the property check is performed right after each emulation step (line 39), so that a violation of ϕ in $\mathbb{S}$ will immediately cause an assertion failure in $\mathbb{P}$.

Note that the loop in the `main` function of $\mathbb{P}$ models all interleavings of actions that can be performed by any agent by means of two nondeterministic assignments (lines 32–34 in Listing 1). In turn, actions (rather, their emulation functions) are enabled based on assumption statements modeling their entry condition (e.g., line 4) which prune away all unfeasible executions. The problem, then, is to show that the flow of actions for the individual agents of $\mathbb{S}$ is preserved in $\mathbb{P}$. In particular, we wish $\mathbb{P}$ to reproduce any possible trace of $\mathbb{S}$ without introducing spurious executions. Intuitively, this is guaranteed in $\mathbb{P}$ by the entry guards and the exit assignments within the emulation functions: the entry guards restrict the possible emulation functions that can be invoked by the scheduler at the current emulation step; the exit assignments update the set of emulation functions from which the scheduler can nondeterministically pick at the next emulation step. For example, considering Fig. 1(a), right after executing action a, nondeterministically either b or c can be executed.

Lemma 3.7 (Completeness of Sequential Emulation). *For each feasible execution f of $\mathbb{S}$, there exists a feasible execution g of $\mathbb{P}$ such that in g the emulation functions are invoked exactly in the same order as their corresponding actions in f.*

Proof. The proof is by induction on the length of the execution trace f of $\mathbb{S}$. In the following, we assume for simplicity that $\mathbb{S}$ is composed of a single agent.

In the `init()` function (lines 12–15), we initialise the program counter of $\mathbb{P}$ so that only the emulation functions for the *initial* actions of $\mathbb{S}$ are enabled, and thus only these functions can be

executed in the first iteration of the scheduler (Section 3.2). Thus, the lemma holds for all traces of length 1.

Now, assuming that $\mathbb{P}$ correctly emulates $\mathbb{S}$ up to $n-1$ actions, let β be the n-th action for some trace of $\mathbb{S}$, and α the action immediately preceding β. We need to show that there exists an execution of $\mathbb{P}$ such that the $(n-1)$-th and n-th emulation steps invoke their emulation functions $\mathtt{f}_\alpha$ and $\mathtt{f}_\beta$, respectively. By the inductive hypothesis, $\mathbb{P}$ emulates $\mathbb{S}$ up to $n-1$ actions, therefore there must exist an execution of $\mathbb{P}$ where $\mathtt{f}_\alpha$ is invoked at the $(n-1)$-th emulation step.

If β immediately follows α within a sequential composition in $\mathbb{S}$, the exit assignment of the program counter at the end of $\mathtt{f}_\alpha$ will match the guard at the beginning of $\mathtt{f}_\beta$, and thus $\mathtt{f}_\beta$ may be invoked right after emulating α.

If β and α occur as parallel actions in $\mathbb{S}$, then β might have been performed instead of α as the $(n-1)$-th action for some other trace of $\mathbb{S}$. In that case, by inductive hypothesis, $\mathtt{f}_\beta$ would have been invoked instead of $\mathtt{f}_\alpha$. This means that the guard for $\mathtt{f}_\beta$ is already satisfied at the $(n-1)$-th iteration of the scheduler. The exit assignments and entry guards within f_α and f_β in $\mathbb{P}$ work on different elements of the program counter, and thus cannot interfere with each other (Fig. 1(b)). Therefore, $\mathtt{f}_\beta$ can be executed right after $\mathtt{f}_\alpha$.

□

Lemma 3.8 (Soundness of Sequential Emulation). *For each feasible execution g of $\mathbb{P}$, there exists a feasible execution f of $\mathbb{S}$ such that the actions in f follow the same order in which the emulation functions are invoked in g.*

Proof. If g has length 1, then it is composed by a call to an emulation function. By construction of the intermediate representation $\mathbb{T}$, and of the start triple in particular, this function must necessarily correspond to an initial action of $\mathbb{S}$. Thus, all executions of length 1 satisfy the lemma. Let us now assume by way of contradiction that there exists an execution of $\mathbb{P}$ where $\mathtt{f}_\gamma$ is invoked at the n-th emulation step right after $\mathtt{f}_\alpha$, but γ never follows α in any trace of $\mathbb{S}$. Assume, again, that $\mathbb{P}$ correctly emulates $\mathbb{S}$ up to the first $n-1$ actions (inductive hypothesis). If $\mathtt{f}_\gamma$ is called at the n-th iteration of the scheduler, then its guard must be satisfied. Two cases may apply: either the guard was also satisfied at the previous emulation step, or it was not.

In the first case, there exists another execution of $\mathbb{P}$ where $\mathtt{f}_\gamma$ is the $(n-1)$-th called function. By the inductive hypothesis, γ must be the $(n-1)$-th action for some trace of $\mathbb{S}$. If α and γ can both be the $(n-1)$-th action, either they are parallel actions or they occur within a nondeterministic choice in $\mathbb{S}$. If they are parallel actions, then we have found a contradiction: there must be a trace of $\mathbb{S}$ where γ follows α. On the other hand, if there is a nondeterministic choice between α and γ, then the exit assignment in $\mathtt{f}_\alpha$ is constructed so that the entry guard of $\mathtt{f}_\gamma$ can never be satisfied (Fig. 1(a)). Thus there cannot be any execution of $\mathbb{P}$ where $\mathtt{f}_\alpha$ is followed by $\mathtt{f}_\gamma$.

The second case implies that an exit assignment at the end of $\mathtt{f}_\alpha$ causes the guard of $\mathtt{f}_\gamma$ to be satisfied. To do that, the exit assignment of $\mathtt{f}_\alpha$ must take into account the guard of $\mathtt{f}_\gamma$. However, this only happens when α is the last action of some sub-process of $\mathbb{S}$, which in turn is sequentially composed with some other sub-process that has γ as a potential initial action. But then there must exist a trace where γ follows α. □

Theorem 3.9 (Correctness of Sequential Emulation of Invariant Properties). *Assuming that ϕ is an invariant property, given the system specification $\mathbb{S}$ and the property ϕ, program $\mathbb{P}$ contains a reachable assertion failure if and only if ϕ does not hold in $\mathbb{S}$, i.e., there exists a feasible execution trace of $\mathbb{S}$ that violates ϕ.*

Proof. The theorem directly follows from Lemma 3.7 and Lemma 3.8.

□

Table 7. Simplified assignment operations to the stigmergy and the environment.

$$\frac{P \xmapsto{x \leftsquigarrow v} P' \quad t = tod() \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I, L \oplus (x, v, t), P', Zc, Zp \cup \{x\}\rangle} \ (\textsc{lstig})$$

$$\frac{P \xmapsto{x \Leftarrow v} P' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{x \Leftarrow v} \langle I, L, P', Zc, Zp\rangle} \ (\textsc{env}_1) \qquad \frac{a \in \mathcal{A} \quad a \xmapsto{x \Leftarrow v} a'}{\langle E, \mathcal{A}\rangle \rightarrow \langle E[x \mapsto v], \mathcal{A}[a'/a]\rangle} \ (\textsc{env}_2)$$

4 EMULATING LABS SPECIFICATIONS

In this section, we present an experimental evaluation that we conducted with a prototype tool that implements the technique from Section 3 to automatically encode LAbS system specifications as sequential C programs, and re-uses off-the-shelf tools developed for the C language in a black-box fashion to carry out the analysis.

We chose C as the target language due to the abundance of mature tools implementing reachability and termination analysis techniques, such as symbolic execution [55], bounded model checking [11], explicit-value analysis [10], predicate-based abstraction [8], k-induction [95], property directed reachability [16, 32, 57], and interprocedural analysis [88]. We use our prototype tool to analyse the selected systems with modern implementations of the above techniques, and report our findings.

Modelling complex SOS rules. To model the complex SOS rules described in Section 2.3, in the emulation program we add a few global variables to model the extended state of each agent, i.e., $\langle I, L, Zc, Zp\rangle$. In particular, we represent each state variable of $\mathbb{S}$ as a global array in $\mathbb{P}$ indexed on the identifier of the agent, to conveniently arrange the states of the different agents.

We then introduce, for each SOS rule, a system-level emulation function whose entry condition leaves the program counter totally unconstrained, so that system-level actions can interleave with the actions of the agents at any time. To prevent spurious triggering of a rule, we add to the entry condition additional constructs that model exactly the premises of the rule. In the emulation function, these constructs are naturally expanded into appropriate assumptions on the newly added global variables. The body of the emulation function will mimic the SOS rule by appropriately manipulating the global variables of the emulation program as prescribed by the rule.

It is worth to recall that providing the emulation functions for each SOS rule is the only manual effort required by our prototype implementation. This is a one-time effort: once the code fragments have been provided for a given language, the procedure can automatically generate emulation programs for every specifications written in that language. In principle, the user may exploit any construct or data structure supported by the target language to describe the state of the system; in practice, however, these choices may affect the feasibility of the analysis. In the context of C, for instance, we have avoided pointers and dynamic memory allocation, in order to obtain programs whose analysis is supported by most verification tools and could be faster to analyse.

Modelling assignment actions to stigmergic variables. Now that the SOS rules for stigmergic interaction are put in place, we are able to introduce the LAbS semantic rule for update actions on stigmergic variables, i.e., $variable \leftsquigarrow value$, and discuss a fragment of C code that implements it.

The action is formalised by rule (LSTIG) in Table 7, where for conciseness we report a simplified version that only allows assignment of values rather than complex expressions. In the rule, $tod()$ represents the timestamp for the new value. The effect of the action is the insertion of a value and

Listing 2. Implementation of rule LSTIG in C.

```
1  // Global variables
2  bool Zp[N][vars], Zc[N][vars];
3  int Zp_count[N], Zc_count[N];
4  int L[N][vars], TimeOf[N][vars];
5
6  // Implement tod()
7  int _tod = 0;
8  int tod() {
9    return _tod++;
10 }
11
12 // Code fragment for rule LSTIG
13 // Check Zc = Zp = ∅
14 assume(Zp_count[tid] == 0);
15 assume(Zc_count[tid] == 0);
16 // Implement L ⊕ (x, v, t)
17 int t = tod();
18 L[tid][x] = v;
19 TimeOf[tid][x] = t;
20 // Implement Zp ∪ {x}
21 Zp[tid][x] = 1;
22 Zp_count[tid]++;
```

its timestamp in the local stigmergy of the agent (denoted by $L \oplus (x, v, t)$), so that the stigmergic propagation rules discussed earlier will asynchronously propagate the value across the local stigmergies of the other agents. The variable is then added to the set Zp of variables to propagate.

Observe that all agent-level rules are guarded by the condition $Zc = Zp = \emptyset$, meaning that an agent has to propagate or confirm all pending variables before continuing its execution.

In the C implementation of the rule, shown in Listing 2, we use `N` and `vars` to denote the number of agents and of stigmergy variables, respectively. We use several global variables to track the state related to these rules. Two arrays of Booleans represent Zc and Zp for each agent: for convenience, we store the number of elements in these sets into two arrays of integers `Zp_count`, `Zc_count`. Lastly, two arrays `L` and `TimeOf` store the value and time for every stigmergy variable of each agent. All these arrays have a statically-determined size. We also implement *tod*() as a function that increments a value from a global variable and returns the result.

The code fragment implementing the rule is parameterised in the identifier `tid` of the agent performing the operation. First, we encode the premise $Zc = Zp = \emptyset$ by means of two assumption statements over the counters `Zp_count[tid]`, `Zc_count[tid]`. If either condition is violated, the emulation function cannot proceed. Otherwise, we encode $L \oplus (x, v, t)$ by assignments to `L` and `TimeOf`, and finally implement the insertion of x into Zp by altering the arrays `Zp` and `Zp_count`. This fragment will be injected into the emulation functions of all triples whose action is a stigmergy assignment (after replacing x and v with the actual variable and value in the assignment).

Modelling environment update actions. This language provides the notion of *environment*, i.e., a data store shared among the agents and operated in a shared-memory fashion. This is formalised by (a simplified version of) rule (ENV_1) in Table 7, which is similar to rule for stigmergic update actions just seen, and lets an agent signal its willingness to assign a value to a shared variable. Then, rule (ENV_2) states that a system composed by an environment E and a set of agents $\mathcal{A}$, such that some agent a is willing to perform an environment update and become a', may evolve to a new system where E and a are replaced by the updated environment and a', respectively.

4.1 Prototype Implementation

In our prototype we use a machine-readable specification language based on LAbS. Listing 3 shows an elementary model of flocking behaviour expressed in this language. External parameters identified with a leading underscore (`_birds`, `_size`, and `_delta`) represent the number of agents, the size of the arena, and the visibility range of an agent, respectively (line 2). Each agent has an

Listing 3. Specification of flocking behaviour.

```
1  system {
2      extern = _birds, _size, _delta
3      spawn = Bird: _birds
4  }
5
6  stigmergy Alignment {
7      link = ((x of c1 - x of c2) * (x of c1 - x of c2)) +
8             ((y of c1 - y of c2) * (y of c1 - y of c2))
9             <= _delta * _delta
10
11     dirx, diry: {-1,1}, {-1,1}
12 }
13
14 agent Bird {
15     interface = x: [0.._size], y: [0.._size]
16     stigmergies = Alignment
17     Behavior = Move; Behavior
18     Move = x, y <- (x + dirx) % _size, (y + diry) % _size
19 }
20
21 check {
22     Consensus = eventually forall Bird b1, forall Bird b2,
23                 dirx of b1 = dirx of b2 and diry of b1 = diry of b2
24 }
```

initial position and direction of movement. The initial position is stored in two variables `x`, `y` that are non-deterministically initialized to a value between `0` and `_size-1` (line 15). The direction of movement is recorded as a pair of values `dirx`, `diry` (line 11) that can be either `-1` or `+1`, depending on whether the bird is moving negatively or positively along either axes (for example `-1,+1` means that the bird is headed left-top). The agents simply keep moving along their current direction (lines 17–18). The direction variables `dirx`, `diry` are stigmergic, and their propagation is controlled by the `link` predicate at line 7. Specifically, when the distance between two birds is `_delta` or less, they will asynchronously agree to move in the same direction. In this example we aim at checking that all birds eventually have the same direction, regardless of their initial direction and position, i.e., whether every pair of birds in the system has the same values for `dirx` and `diry` (line 22).

Our prototype takes as input the system specifications in the format described above, the external parameters (if any), the scheduling policy to enforce (either full interleaving or round-robin), and the name of the verification tool to use as a back end for the analysis.

After parsing the input specification, the tool generates the triple structure (Sect. 3.1), and then the sequential program (Sect. 3.2). The program is then prepared for either reachability or termination analysis. Depending on the property of interest, one of the statements at line 18 and line 19 of Listing 1 is removed. For invariant properties, reachability of the `error` statement (Listing 1, line 18) will indicate that the property does not hold. For emergent properties, successful termination of the program (Listing 1, line 19) will imply satisfaction of the property. Similarly, one of the two branches (Listing 1, line 23) is removed, depending on the scheduling policy. The program is then

instrumented for the specific analysis back end by introducing the correct primitives for non-deterministic initialisation of variables, assertions, and assumptions. The instrumented program is eventually fed to the back end to carry out the analysis.

4.2 Case Studies

We now describe our selection of systems used for our experimental evaluation.

`Flock` is the example introduced in Section 2.3, and specified in Listing 3. This system is a simplified version of the *boids* model [89], that implements only the *alignment* rule. Agents are placed in a two-dimensional arena and start from a chaotic state of motion: when they are sufficiently close, they may agree to move in the same direction. We would like to make sure that all the agents eventually move in one direction. In our experiments we have used the following values for external parameters: (`_birds` = 3, `_size` = 5, `_delta` = 5). For this system we assume round-robin scheduling.

`Boids` extends the `flock` example with a cohesion mechanism that makes agents move closer to each other [89]. We add separate stigmergic variables to represent the *group leader*, the *group size*, and the position of the leader. Each agent is initially the leader of its own group, which has size one; however, an agent may become a *follower* of another group which is at least as large as the one it belongs to. If so, the agent updates the size count of that group to include itself. Leaders also store their position in the stigmergy: a follower that is too distant from its leader may decide to move towards it. As a consequence of the additional rules, this system is considerably more complex than `flock`. We are interested in checking whether all agents eventually reach a consensus on the leader after an arbitrary number of transitions. The parameters for this system are the same as in `flock`. We assume round-robin scheduling.

`Formation` describes a system of N agents placed on a segment of length L. They start from non-deterministically-chosen positions on the segment and must move so that their distance from each other is eventually δ or more. To achieve this goal, all agents repeatedly write their id on two variables *left*, *right*. Due to the link predicates, these values may only be changed by other agents that are closer than δ. So, for instance, when an agent notices that *left* is different from its own id, it makes a step to the right (and vice versa), unless it is at either position 0 or $L - 1$. We instantiate the specifications with parameters ($L = 10, N = 3, \delta = 2$). We wish to check that no agent will ever be in a position outside of the range $[0, L - 1]$, and also that all the agents eventually are at least at a distance δ from each other. For this system we assume round-robin scheduling.

`Approx` is an approximate majority population protocol presented in [2]. Each agent has an initial opinion, encoded as either 0 or 1. Let n and y the number of agents with initial opinion set to 0 or 1, respectively. The aim is to reach a state where all agents have the opinion that initially has the majority. We analyse a three-agent system `approx-a` ($y = 1, n = 2$) and a five-agent one, which we call `approx-b` ($y = 2, n = 3$). In both systems, y is initially the minority opinion. A safety property of interest is that the system should never reach a state where all agents agree on y.

`Maj` is another population protocol where agents should eventually have the opinion that had the majority in the initial state. The properties of interest are the same as those of the `approx` protocol. However, this one has been proven correct for any initial configuration [3]. Automated tools for the analysis of population protocols have also been able to verify it [12]. As with the other population protocol, we analyse this system with parameters ($y = 1, n = 2$), and are interested in checking that a state where all agents agree on y is never reachable.

4.3 Experimental Results

For the experiments we considered the following categories of techniques and tools. We tried Symbiotic [18], which combines symbolic execution with program slicing to perform both reachability and

termination analysis. For bounded model checking, we tried SAT-based bounded model checking with CBMC [21], and SMT-based bounded model checking with ESBMC [36], and SMACK [87]. We also consider a competition release [7] of CBMC, which performs termination analysis by repeatedly unrolling all loops with an increasing bound and using BMC to check unwinding assertions. An unwinding assertion is an assertion statement that is placed after the k-th iteration of an unwound loop, and is satisfied if and only if the unsound loop never performs more than k iterations. If all unwinding assertions are satisfied for a given bound k, then all loops have a bounded number of iterations and thus the program unconditionally terminates. For abstraction-based techniques, we experimented with summary-based interprocedural analysis based on different abstract domains as implemented in 2LS [93]. We tried explicit-value analysis and a CEGAR loop based on predicate abstraction with CPAchecker [9]. We also used an automata-based CEGAR loop implemented by Ultimate Automizer [47]. As for inductive techniques, we evaluated the implementations of k-induction [95] provided by 2LS [19], CPAchecker, and ESBMC. We also experimented with more recent algorithms based on property directed reachability (PDR): RecMC [57] as implemented by Seahorn [45], and the IC3 [16] implementation of VVT [44].

In a preliminary experimental phase, we also tried other tools, namely: AProve [39] (which performs termination analysis based on symbolic execution), IKOS [17] (abstract interpretation), and Kratos [20] (lazy predicate abstraction). However, these tools did not produce any conclusive results on our programs and thus we did not consider them for the final selection of verification tasks. Specifically, AProve always returns a *maybe* verdict; IKOS declares that the safety of the benchmarks cannot be conclusively proven and quits; lastly, all Kratos benchmarks hit the time limit. It may be interesting to investigate whether future versions of these tools would be able to provide conclusive verdicts for our emulation programs.

All the experiments were performed on a dedicated 64-bit GNU/Linux workstation with kernel 4.9.95, equipped with 128GB of physical memory and a dual 3.10GHz Xeon E5-2687W 8-core processor. We set a time limit of 12 hours and a memory limit of 32 GB for the analysis.

Tables 8 and 9 report our experimental results in invariance (reachability) and emergence (termination) analysis of C emulation programs. In both tables, the top row and the leftmost column refer to the considered multi-agent system and the program analysis technique (along with the specific implementation), respectively. For each system, we include a reference to existing work describing it, and to previous verification results whenever possible.

We list the different techniques from top to bottom according to the following categories. For Table 8 the categories are: symbolic execution, bounded model checking, abstraction-based techniques, and induction-based techniques. For Table 9 we consider symbolic execution, bounded model checking with completeness threshold detection, and interprocedural analysis based on summarization. The bottom of the table reports the verdict we were able to draw by only inspecting our experimental results, without exploiting any previous knowledge of the benchmarks. In the internal cells of the two tables, we report the partial verdicts along with the decision time (in minutes) for each tool and system. Conclusive results are marked with ✓ or ✗ to respectively denote that the property under analysis was successfully verified or violated. Superscripts provide further details on the inconclusive experiments.

Main insights. By looking at the separate columns of both Tables 8 and 9, we observe that for each of the systems under consideration at least one tool is able to generate a conclusive verification verdict, and that the conclusive verdicts for a given system are always consistent. Therefore, we can confidently draw a conclusive verification verdict for every system.

Interestingly, our verdicts do confirm all the known results from the literature, specifically, for (`approx-a`, `approx-b`, and `maj`) we can confirm the findings in [3, 12]. Even more interestingly, we

Table 8. Analysis of invariant properties (reachability) $-^a$: Timeout (12 hours). $-^b$: Inconclusive analysis reported by the tool. $-^c$: Out of memory (32 GB). $-^*$: The tool requires an array-free encoding.

	`formation`	`approx-a`	`approx-b`	`maj`
Symbolic execution (Symbiotic)	0.01 ✓	206.83 ✗	60.47 ✗	$-^a$
Bit-precise BMC (CBMC)	$-^a$	0.01 ✗	0.01 ✗	$-^a$
Word-level BMC (ESBMC)	$-^a$	0.08 ✗	0.07 ✗	$-^a$
Word-level BMC (SMACK)	$-^a$	0.67 ✗	1.83 ✗	$-^a$
Explicit-value analysis (CPAchecker)*	$-^c$	337.08^b	$-^c$	62.37 ✓
Predicate abstraction+CEGAR (CPAchecker)*	0.34 ✓	0.08 ✗	0.03 ✗	$-^c$
Automata+CEGAR (Automizer)	5.98 ✓	1.17 ✗	45.7 ✗	$-^a$
k-induction (CPAchecker)*	$-^c$	0.1 ✗	0.05 ✗	$-^c$
k-induction (2LS)*	0.05 ✓	0.01 ✗	0.01 ✗	$-^a$
k-induction (ESBMC)	0.01 ✓	0.01 ✗	0.05 ✗	0.01 ✓
PDR (Seahorn)	0.3 ✓	0.03 ✗	0.5 ✗	4.67 ✓
PDR (VVT)	0.03 ✓	0.01 ✗	0.01 ✗	0.01 ✓
	✓	✗	✗	✓

manage to successfully verify a liveness property for `boids`. It is worth to observe that this requires analysis of the behaviour of the flock up to an unbounded number of steps, which is particularly challenging. Besides simulation [76], we are only aware of one previous successful attempt to analyse this system in the bounded case [26]. We are not aware of previous attempts of unbounded verification of this system.

Analysis of invariant properties. As shown in Table 8, symbolic execution [55] gives correct results but seems to have issues with performance. The analysis of both `approx-a` and `approx-b` takes three and one hour, respectively, and the tool is not able to verify `maj` due to timeout. Bounded model checking [11] is consistently quick in detecting property violations, with either SAT or SMT decision procedures, and all the considered tools for this category were able to generate precise violation witnesses. However, this technique alone cannot provide conclusive results in the absence of property violations, such as `formation` and `maj`. For these systems, we repeatedly increased the verification bound until timing out, to double-check the consistency with the other approaches.

Abstraction-based analysers [10, 40, 47] seem to complement these limitations, as they can successfully determine the safety for `formation` and `maj` in half of the cases. In contrast, the results on the unsafe instances (confidently claimed as such via bounded analysis) are sometimes inconclusive. Analysis procedures based on CEGAR loops [22] correctly identify the `approx` programs as unsafe and the `formation` program as safe, but they run into memory or time limits issues when considering the more complex `maj` program. This confirms once again that under- and over-approximation are orthogonal to each other.

Table 9. Analysis of emergent properties (termination). $-^a$: Timeout (12 hours). $-^b$: Inconclusive analysis reported by the tool. $-^c$: Out of memory (32 GB). $-^*$: The tool requires an array-free encoding.

	`formation`	`flock`	`boids`
Symbolic execution (Symbiotic)	488.40^b	$-^a$	$-^a$
BMC+completeness threshold (CBMC)	214.32 ✓	243.84 ✓	49.52 ✓
Summarization+intervals (2LS)*	0.08^b	0.03^b	32.15^b
Summarization+equalities (2LS)*	319.40^b	107.72 ✓	$-^a$
	✓	✓	✓

Listing 4. Counterexample for the `approx-b` system.

```
<initialization>
No 0:   state <- 0
No 1:   state <- 0
No 2:   state <- 0
Yes 3:  state <- 0
Yes 4:  state <- 0
        agent <-- -128
        message <-- -128
No 0:   state <- 0
No 1:   state <- 0
No 2:   state <- 0
Yes 3:  state <- 1
Yes 4:  state <- 1
<end initialization>
No 1:   agent <-- 1
No 1:   message <-- 0
Yes 3:  state <- 2
Yes 4:  agent <-- 4
Yes 4:  message <-- 1
No 2:   state <- 2
Yes 3:  state <- 1
No 1:   state <- 2
No 1:   state <- 1
No 2:   state <- 1
Yes 4:  agent <-- 4
Yes 4:  message <-- 1
No 2:   agent <-- 2
No 2:   message <-- 1
No 0:   state <- 2
No 2:   agent <-- 2
No 2:   message <-- 1
No 0:   state <- 1
<property violated: 'NeverYConsensus'>
```

Interestingly, we observe the superiority of inductive techniques, i.e., k-induction[3] [95] and property directed reachability [16], over the other approaches we considered. These techniques exhibit outstanding performances with consistent verdicts; produce precise witnesses for violated properties, with comparable performances to the fastest bounded model checker; and are competitive, if not superior, to abstraction-based tools on safe systems.

Analysis of emergent properties. In Table 9 we observe overall less data points as well as a smaller portion of conclusive verdicts with respect to Table 8. In fact, the presence of non-linear operations appears to be a major hindrance for the tools that we have considered for termination analysis. This issue is related to the specific language considered in our experimental evaluation, as non-linearity stems directly from the operational semantics.

Nevertheless, we do manage to find at least one conclusive verdict for each problem. In particular, 2LS can confirm the termination of `flock` when using the *equalities* abstract domain, while bounded model checking with completeness detection is able to verify the termination for all three systems under verification, including `boids`.

[3]As implemented by ESBMC 6, which also infers invariants through interval analysis. As shown in Table 8, the other implementations that we have considered (CPAchecker and 2LS) sometimes report inconclusive results on safe systems.

Translating CBMC error traces. We have instantiated the counterexample translation procedure described in Section 3.2 to automatically translate the error traces from CBMC into a readable LAbS-like syntax. Listing 4 shows a translated counterexample for system `approx-b`, which reports the initial state of the system (lines **??**–**??**), followed by a sequence of assignments performed by the agents that leads to a property violation (lines **??**–**??**, where `Yes` and `No` refer to the initial opinion of the agent performing the action). Supporting other verifiers would take some effort, as it requires ad-hoc parsing for each specific counterexample format. In that respect, we plan to support counterexample translation from standardised error witness formats, e.g., [6], which would make it possible to translate counterexamples from many program verifiers with a unique procedure.

Availability of artifacts. We generated all emulation programs by using the automated tool SLiVER (version 1.5), which implements the mechanised translation procedure described in Section 3. A Linux release of the tool is available online.[4] We have also published a persistent replication package [29] containing SLiVER, as well as the LAbS specifications, the emulation programs, the full output produced by each verification tool for every task, and instructions to reproduce our experiments.

5 EMULATING OTHER FORMALISMS

To further illustrate the applicability of our approach, we briefly describe another use case, namely *service choreographies*. A choreography is a global description of the interactions that two or more *peers* should perform in order to achieve a common goal. Interestingly, the behaviour of peers may be synthesized from the choreography itself, by means of *projection* procedures. An example is *natural projection*, which essentially amounts to hiding (i.e., renaming to an invisible action τ) all those actions that the peer cannot perform. Thus, it is useful to determine whether a given choreography is implemented by a set of synthesized peers. This is known as the problem of *realizability* (under the given projection operation). The overall problem of realizability usually implies checking some behavioural equivalence (e.g., trace equivalence) between the choreography and the implementation, and additionally verifying that the implementation is deadlock-free. While equivalence checking is currently out of our scope, we will demonstrate that we can already check deadlock-freedom on choreography implementations.

5.1 Chor, a Choreography Description Language

In this work, we focus on a subset of Chor [86], a choreography description language with a formal syntax and semantics.[5] A Chor specification is composed of *local actions* and *communications*. A local action α^x denotes an activity that the peer x may perform asynchronously, while a communication $\alpha^{[x,y]}$ denotes an action that two peers x, y must synchronise upon. Specifications may also contain the invisible action τ. Smaller choreographies may be composed into larger ones by means of operators such as choice (+), sequential composition (;), or parallel composition ($\|$). Additionally, Chor provides a *loop* operator $*P$ that is equivalent to the (infinitely-branching) choice $\tau + P + (P;P) + (P;P;P) + \ldots$.

Chor is equipped with a trace semantics, and the problem of realizability reduces to verifying trace equivalence between the choreography and its implementation, i.e., the parallel composition of peers obtained through natural projection. Additionally, the implementation is required to be deadlock-free, meaning that all peers must successfully terminate in every execution. In fact, even

[4] https://github.com/labs-lang/sliver

[5] To maintain a uniform notation, in this work we will use τ and + to denote Chor's internal action and choice operator, respectively. The original Chor syntax uses *skip* and $\sqcap$, instead.

Listing 5. An example Chor choreography and its natural projections.

$$
\begin{aligned}
\textsc{Stock} &\triangleq (iron^{bk} + steel^{bk}); *(look^{bk}); bid^{[bk,mk]}; (save^{mk} \| check^{mk}); \\
&\quad result^{[bd,mk]}; change^{bd}; (notify^{[bd,bk]} + \tau) \\
\textsc{Board} &\triangleq result^{[bd,mk]}; change^{bd}; (notify^{[bd,bk]} + \tau) \\
\textsc{Broker} &\triangleq (iron^{bk} + steel^{bk}); *(look^{bk}); bid^{[bk,mk]}; (notify^{[bd,bk]} + \tau) \\
\textsc{Market} &\triangleq bid^{[bk,mk]}; (save^{mk} \| check^{mk}); result^{[bd,mk]} \\
&\quad *
\end{aligned}
$$

though the implementation may be trace equivalent to its choreography, some executions may leave one or more peers in a deadlocked state where they are still willing, but unable, to interact with peers which have terminated.

5.2 Verifying Deadlock Freedom in Choreography Implementations

In general, an emulation program deadlocks if and only if it reaches a state from which no emulation function is enabled. This, in turn, may happen if and only if the original system is not deadlock-free. Thus, we may prove deadlock freedom by checking that the scheduler is able to invoke at least one emulation function at every iteration.

As an example, let us consider a choreography Stock that models a metal stock market made of three peers (the *market* itself, a *broker*, and an announcement *board*) [92]. The choreography is shown in Listing 5, where we use *mk*, *bk*, and *bd* as shorthands for *market*, *broker*, and *board*, respectively. In the choreography, the broker decides to buy either *steel* or *metal*, and then *look*s until a sale becomes available. When this happens, the broker sends a *bid* to the market. Upon receiving the bid, the market *save*s it in a database and *check*s whether it is higher than the best one so far (these two action may be performed in any order). Then, it sends the *result* of the check to the announcement board. Finally, the board can either *change* its content and *notify* the broker (if their bet is now the best one), or do nothing (τ).

Listing 5 also shows the behaviour of each peer, obtained through natural projection. To verify whether Market||Broker||Board is deadlock-free, we encode it first as a triple structure (shown in Fig. 5) and then as an emulation program. The structure of the latter resembles the one shown in Listing 1. This time, the `check()` function contains an assertion that is satisfied if and only if there is at least one enabled emulation function. Since the choreography is stateless, emulation functions are empty and only update the program counter.

2LS is able to find a violation witness in about 4 seconds. This result confirms the finding in [92]: namely, `Board` and `Broker` may take different branches, allowing the latter to remain stuck on a *notify* communication.

We also checked a safe variant of the same choreography, in which `Board` and `Broker` will always perform *notify* before terminating:

$$
\begin{aligned}
\textsc{Stock-Safe} &\triangleq (iron^{bk} + steel^{bk}); *(look^{bk}); bid^{[bk,mk]}; (save^{mk} \| check^{mk}); \\
&\quad result^{[bd,mk]}; (change^{bd} + \tau); notify^{[bd,bk]}
\end{aligned}
$$

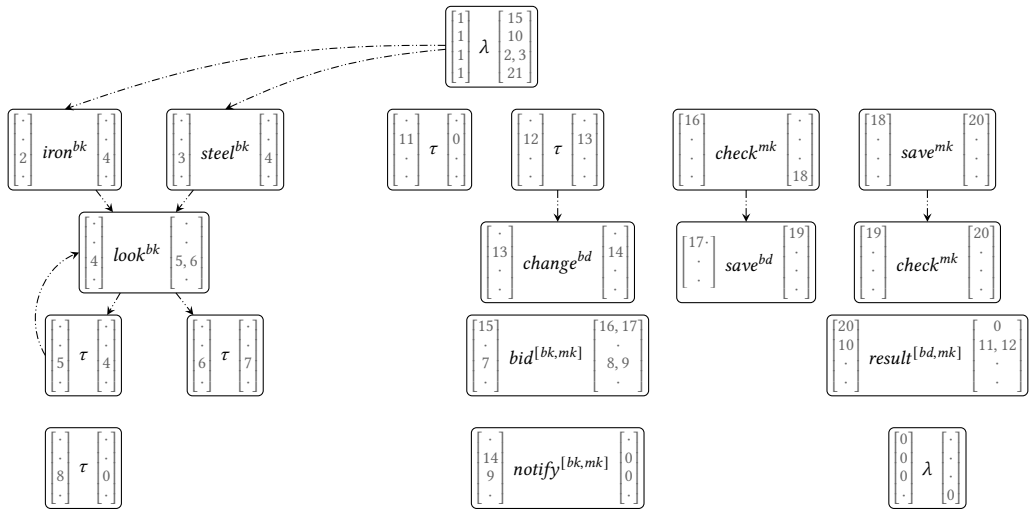


Fig. 5. Triple structure encoding the implementation of the Stock choreography.

This variation is deadlock-free, and indeed 2LS can successfully verify its emulation program in about 5 seconds. As with the other experiments in this paper, these results may be reproduced by means of our replication package [29].

6 RELATED WORK

Several semantics-based verification techniques have been recently proposed. Most notably, K is an integrated framework for semantics-based language specification, execution, and formal analysis of programs. In principle, in K one can define syntax and operational semantics of any language, and can automatically instantiate a reachability verifier for programs for his language [96]. Naturally, this comes at the cost of developing the full operational semantics of the language. Moreover, K requires rewrite-based semantics rules, so it might be necessary to adapt the existing semantics of the source language in that sense.

Our approach is also semantics-based, but focusses on quick prototyping of verification flows for concurrent and distributed systems. We assume a predefined set of composition operators, but no specific format for the semantics rules: one can model any semantics as long as there is a way of doing so in the target language. The encoding of the single rules is currently done manually, but could be mechanised by adopting a machine-tractable format for the semantics.

On the analysis side, K mainly relies on symbolic execution, and there appears to be an ongoing effort[6] to translate K's underlying specification language (Kore) into the LLVM intermediate representation, which could allow to reuse LLVM-oriented verification tools. In contrast, our encoding into sequential imperative languages allows to immediately inherit a large collection of analysis techniques (from traditional symbolic execution to inductive methods) and off-the-shelf verifiers, including different LLVM-based implementations. Moreover, our encoding is a sequential program, which allows for instance to use techniques for the analysis of sequential programs, broadening the set of candidate verifiers even further.

Other semantics-based approaches include generation of verification conditions in the form of Horn clauses [24], termination analysis [102], simulation [75], or model checking through rewriting

[6]https://github.com/kframework/llvm-backend

systems [90]. With the exception of [90], all the approaches mentioned above rely on a structural operational semantics (SOS) [83] of the source language, like ours; however, all of them are tied to a specific analysis technique. In contrast, our approach is modular with respect to the back end. Any verification tool can be used as a black box, as long as it can handle the very simple syntax of our encoding (essentially just arrays and loops), and support the standard program instrumentation primitives for non-deterministic initialisation, assertions, and assumptions. For instance, we also used triple structures to implement a translation [30] from LAbS specifications into LNT programs [38]. On one hand, the translation into LNT may allow to verify a larger set of properties than simple invariance or emergence, by means of the CADP analysis toolbox [37] and its expressive property language. On the other hand, the symbolic C-based encoding presented in this paper allows to immediately inherit a comprehensive range of mature and sophisticated verification techniques developed for mainstream programming languages. Being semantics-based, this approach may be applied to analyse different specification languages other than CCS and LAbS. For instance, one could generate emulation programs from AbC specifications [1] with a procedure similar to ours, by means of an intermediate representation based on guarded transitions [27]. It is also worth to add that the emulation program generated by our encoding is sequential, which allows to use tools that do not support concurrency.

Different languages and frameworks for the specification and analysis of distributed systems have been put forward. CADP is oriented at concurrent value-passing systems, and can perform explicit-state model checking of temporal properties specified in a variant of the modal μ-calculus, with support for compositional verification [65]. TLA+ [64] supports checking assertion violations and verifying linear temporal properties through explicit-state model checking [106], theorem proving [23], and symbolic model checking [58]. Disel [94] aims at modular verification, i.e., verifying individual components and exploiting these proofs when verifying a larger system. Verification is type-based, relies on theorem proving, and is currently limited to safety properties. Similarly, Ivy [78] allows to check safety properties through interactive theorem proving and modular verification. Differently from Disel, Ivy can verify parameterized protocols with an arbitrary number of participants. As both Ivy and Disel rely on theorem proving, verification is generally an interactive procedure. In our technique, the only requirement for the end user is a sufficient familiarity with the domain-specific language, thus we deliberately target push-button verification technology. Moreover, CADP, TLA+, Disel, and Ivy rely on their own (fixed) specification language. Our approach is parametric with respect to the semantics of the source language.

Use of specific process algebras as a modelling tool has already been considered in the domain of natural systems [80, 97], and also exploited for quantitative verification via probabilistic model checking [60, 67].

Sequentialization [63, 85] was originally introduced with the goal to re-use sequential verification tools for the analysis of concurrent imperative programs. Tailored sequentialization schemas for shared-memory concurrent programs in a C-like syntax [33, 52, 53] focus on context-bounded analysis for an efficient falsification of safety properties. In contrast, our emulation programs can analyse richer properties and up to an infinite number of steps. More recent approaches for message-passing systems [4, 15] adopt specific assumptions on the communications, e.g., bounded number of outbound message, FIFO handling, etc. We do not add any such assumption as in our approach the possible evolutions of the system are only controlled by the operational semantics of the specification language.

MCMAS [68] allows to prove properties for systems of unbounded size with agents interacting in a shared environment, but lacks support for value-passing actions and relies on explicit model checking. AJPF [14] provides a toolkit to perform model-checking on a variety of agent-oriented languages but is tied to a specific, explicit-state verification back end.

Peregrine [12] can verify and simulate population protocols for an unbounded number of agents. Peregrine is only concerned with verifying whether a protocol *computes a predicate* ψ over the initial state of its population. This means that every fair execution of the protocol *stabilizes*, i.e., satisfies $\Diamond\Box\psi'$ for some appropriate predicate ψ'. For instance, one may verify that the `Maj` protocol computes the predicate "a majority of agents has opinion Y" by checking that, eventually, all agents reach a lasting consensus on Y. In contrast, our encoding also supports the verification of arbitrary invariants throughout the population's evolution. While it currently does not allow to check a property $\Diamond\Box\psi'$ directly, it may be used to verify first $\Diamond\psi'$, and then $\psi' \Rightarrow \Box\psi'$.[7] The conjunction of these two properties implies $\Diamond\Box\psi'$.

There is an abundance of both visual and textual choreography description languages, including WS-CDL [103], conversation protocols [35], and variants of the BPMN graphical notation [77]. These formalisms have similar expressive power to Chor and have been formally analysed by resorting to a common intermediate representation [43] or to general-purpose process algebras [54]. Thus, it is reasonable to expect that our approach could be applied to them as well. Simulation is still widely used in the context of multi-agent systems [31, 56, 62, 69, 82, 91] for bug-finding and quick feedback [51], but due the presence of concurrency and asynchrony it may only be marginally effective in many cases [104].

7 CONCLUSION

We have proposed a novel semantics-based technique that reduces property checking of distributed systems to verification of sequential programs. An encoding procedure translates the initial system under consideration into a sequential program, over which reachability or termination analysis can be performed as an alternative to analysing the initial system. An intermediate representation based on sequential programs guarantees separation of concerns between the specificities of the source language and those of the back end verification technology. We have shown how, thanks to our technique, different representative classes of concurrent systems described with an existing formal specification language can be automatically verified by using mature verification tools for C programs as black boxes. Our experimental evaluation, the fact that our approach allows automated verification of Boids up to an unbounded number of steps, and the consistent effectiveness of modern inductive-style methods, encourage us to continue in this direction.

In our view, our contribution is a first step towards fully mechanisable procedures to build new push-button verification tools for domain-specific languages for distributed systems based on their operational semantics. The approach puts the end-user of such tools at ease, as familiarity with the source language is the only usage requirement. The methodology has a broad potential impact, as it can facilitate the adoption of state-of-the-art techniques for computer-aided verification among diverse research communities.

The generality of our technique lies in its ability to translate from any domain-specific language, equipped with a structural operational semantics, to any imperative language with arrays and loops. Currently, the translation procedure still requires a one-time manual effort to render the semantics of the actions of the source language as code fragments to be expanded within the emulation functions stubs in the target program. In general, the complexity of this manual step depends on that of the semantic rules being translated. Basic process algebras like CCS require little effort, while complex domain-specific languages usually need more elaborate emulation programs to deal with advanced features, such as value-passing and asynchronous interactions like in LAbS.

[7] In general, one can always verify that $\psi \Rightarrow \phi$ holds in $\mathbb{S}$, by verifying ϕ against some $\mathbb{S}'$ which is identical to $\mathbb{S}$, except that its initial states are all the states of $\mathbb{S}$ which satisfy ψ.

In general, we could work around the need to manually write templates for emulation functions by synthesizing them from machine-readable semantic rule formats like MSOS [74].

We believe that the set of composition operators we have chosen allows us to support a large variety of source languages. However, formalisms relying on more complex compositions would be harder to fit into our approach. Consider for instance CSP's *interrupt* operator $P \triangle Q$, which defines a process that behaves like P but can be interrupted and start behaving like Q at any moment (unless P terminates, in which case Q is discarded) [50]. Since our procedure is defined by induction on the structure of processes, in order to appropriately generate triples for a process $P \triangle Q$ we would need to extend our definitions of translation and enabler functions (Tables 5 and 6) accordingly. In principle, a suitable machine-readable format to describe the triple structure of a composite process (e.g., $P \triangle Q$) in terms of those of its terms (e.g., P and Q) would allow to extend our encoding to custom composition operators. We leave this for future work.

The current version of our encoding supports simple invariant and emergent properties, respectively expressed in terms of reachability or termination of the sequential emulation program. We are planning additional work to support a wider range of properties. We also plan to expand the experimental evaluation to systems with a dynamic, and possibly unbounded, number of agents. To that end, we will investigate which state-of-the-art techniques and tools can effectively deal with programs that feature dynamic memory allocation. Alternatively, one might think of adapting cutoff techniques developed for parameterised model checking [61].

As large systems may be out of reach, we plan to complement the presented approach with simulation-based analysis, such as statistical model checking [66]. In some cases, it may also be possible to reduce property verification for an unbounded number of transitions to bounded model checking by calculating completeness thresholds [59] by inspecting the structure of the system specifications.

Lastly, we will consider equipping our prototype with tailored back ends aimed at distributing systems analysis over large computational clusters (e.g., by partitioning schedules as in [53]). This would represent a very powerful instrument to gather deep insights on the nature of complex systems. Relating such insights to the other findings from different disciplines and possibly share them with broad scientific audiences [5] would be extremely interesting.

ACKNOWLEDGMENTS

Work partially funded by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems). The authors would like to thank the anonymous reviewers, whose comments and suggestions helped improve the article.

REFERENCES

[1] Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. 2020. Programming interactions in collective adaptive systems by relying on attribute-based communication. *Science of Computer Programming* 192 (2020), 102428. https://doi.org/10.1016/j.scico.2020.102428

[2] Dana Angluin, James Aspnes, and David Eisenstat. 2008. A Simple Population Protocol for Fast Robust Approximate Majority. *Distributed Computing* 21, 2 (2008), 87–102. https://doi.org/10.1007/s00446-008-0059-z

[3] James Aspnes and Eric Ruppert. 2009. An Introduction to Population Protocols. In *Middleware for Network Eccentric and Mobile Applications*, Benoît Garbinato, Hugo Miranda, and Lúıs E. T. Rodrigues (Eds.). Springer, 97–120. https://doi.org/10.1007/978-3-540-89707-1_5

[4] Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. 2017. Verifying distributed programs via canonical sequentialization. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 110:1–110:27. https://doi.org/10.1145/3133934

[5] M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, M. Viale, and V. Zdravkovic. 2008. Interaction Ruling Animal Collective Behavior Depends on Topological Rather than Metric Distance: Evidence from a Field Study. *Proceedings of the National Academy of Sciences* 105, 4

(2008), 1232–1237. https://doi.org/10.1073/pnas.0711437105

[6] Dirk Beyer. 2016. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 9636)*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, 887–904. https://doi.org/10.1007/978-3-662-49674-9_55

[7] Dirk Beyer. 2019. Automatic Verification of C and Java Programs: SV-COMP 2019. In *25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 11429)*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer, 133–155. https://doi.org/10.1007/978-3-030-17502-3_9

[8] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The Software Model Checker BLAST. *Software Tools for Technology Transfer* 9, 5-6 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[9] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *23rd International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[10] Dirk Beyer and Stefan Löwe. 2013. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *16th International Conference on Fundamental Approaches to Software Engineering (FASE) (LNCS, Vol. 7793)*, Vittorio Cortellessa and Dániel Varró (Eds.). Springer, 146–162. https://doi.org/10.1007/978-3-642-37057-1_11

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS) (LNCS, Vol. 1579)*, Rance Cleaveland (Ed.). Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[12] Michael Blondin, Javier Esparza, and Stefan Jaax. 2018. Peregrine: A Tool for the Analysis of Population Protocols. In *30th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 604–611. https://doi.org/10.1007/978-3-319-96145-3_34

[13] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. 1999. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press.

[14] Rafael H. Bordini, Louise A. Dennis, Berndt Farwer, and Michael Fisher. 2008. Automated Verification of Multi-Agent Programs. In *23rd International Conference on Automated Software Engineering (ASE)*. IEEE, 69–78. https://doi.org/10.1109/ASE.2008.17

[15] Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. 2018. On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony. In *30th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 372–391. https://doi.org/10.1007/978-3-319-96142-2_23

[16] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS, Vol. 6538)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[17] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In *12th International Conference on Software Engineering and Formal Methods (SEFM) (LNCS, Vol. 8702)*. Springer, 271–277. https://doi.org/10.1007/978-3-319-10431-7_20

[18] Marek Chalupa, Martina Vitovská, Martin Jonás, Jiri Slaby, and Jan Strejcek. 2017. Symbiotic 4: Beyond Reachability - (Competition Contribution). In *23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 10206)*, Axel Legay and Tiziana Margaria (Eds.). Springer, 385–389. https://doi.org/10.1007/978-3-662-54580-5_28

[19] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. 2015. Synthesising Interprocedural Bit-Precise Termination Proofs. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 53–64. https://doi.org/10.1109/ASE.2015.10

[20] Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, and Marco Roveri. 2011. Kratos - A Software Model Checker for SystemC. In *23rd International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 310–316. https://doi.org/10.1007/978-3-642-22110-1_24

[21] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[22] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[23] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. 2012. TLA+ Proofs. In *18th International Symposium on Formal Methods (FM) (LNCS, Vol. 7436)*, Dimitra Giannakopoulou and Dominique Méry (Eds.). Springer, 147–154. https://doi.org/10.1007/978-3-642-32759-9_14

[24] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2015. Semantics-Based Generation of Verification Conditions by Program Specialization. In *17th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, Moreno Falaschi and Elvira Albert (Eds.). ACM, 91–102. https://doi.org/10.1145/2790449.2790529

[25] Rocco De Nicola. 2011. Process Algebras. In *Encyclopedia of Parallel Computing*, David A. Padua (Ed.). Springer, 1624–1636. https://doi.org/10.1007/978-0-387-09766-4_450

[26] Rocco De Nicola, Luca Di Stefano, and Omar Inverso. 2020. Multi-agent systems with virtual stigmergy. *Science of Computer Programming* 187 (2020), 102345. https://doi.org/10.1016/j.scico.2019.102345

[27] Rocco De Nicola, Tan Duong, and Omar Inverso. 2020. Verifying AbC Specifications via Emulation. In *9th International Symposium on Leveraging Applications of Formal Methods (ISoLA) (LNCS, Vol. 12477)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 261–279. https://doi.org/10.1007/978-3-030-61470-6_16

[28] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. 2017. Combining Model Checking and Runtime Verification for Safe Robotics. In *17th International Conference on Runtime Verification (RV) (LNCS, Vol. 10548)*, Shuvendu Lahiri and Giles Reger (Eds.). Springer, 172–189. https://doi.org/10.1007/978-3-319-67531-2_11

[29] Luca Di Stefano, Rocco De Nicola, and Omar Inverso. 2021. *Replication Package for the paper: Verification of Distributed Systems via Sequential Emulation.* https://doi.org/10.5281/zenodo.5348289

[30] Luca Di Stefano, Frédéric Lang, and Wendelin Serwe. 2020. Combining SLiVER with CADP to Analyze Multi-agent Systems. In *22nd International Conference on Coordination Models and Languages (COORDINATION) (LNCS, Vol. 12134)*, Simon Bliudze and Laura Bocchi (Eds.). Springer, 370–385. https://doi.org/10.1007/978-3-030-50029-0_23

[31] Gilberto Echeverria, Séverin Lemaignan, Arnaud Degroote, Simon Lacroix, Michael Karg, Pierrick Koch, Charles Lesire, and Serge Stinckwich. 2012. Simulating Complex Robotic Scenarios with MORSE. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR) (LNCS, Vol. 7628)*. Springer, 197–208. https://doi.org/10.1007/978-3-642-34327-8_20

[32] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. 2011. Efficient Implementation of Property Directed Reachability. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Per Bjesse and Anna Slobodová (Eds.). FMCAD Inc., 125–134.

[33] Bernd Fischer, Omar Inverso, and Gennaro Parlato. 2013. CSeq: A concurrency pre-processor for sequential C verification tools. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 710–713. https://doi.org/10.1109/ASE.2013.6693139

[34] Wan Fokkink. 2000. *Introduction to Process Algebra.* Springer.

[35] Xiang Fu, Tevfik Bultan, and Jianwen Su. 2004. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theoretical Computer Science* 328, 1-2 (2004), 19–37. https://doi.org/10.1016/j.tcs.2004.07.004

[36] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. 2018. ESBMC 5.0: An Industrial-Strength C Model Checker. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 888–891. https://doi.org/10.1145/3238147.3240481

[37] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. 2011. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 6605)*. Springer, 372–387. https://doi.org/10.1007/978-3-642-19835-9_33

[38] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. 2017. From LOTOS to LNT. In *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday (LNCS, Vol. 10500)*. Springer, 3–26. https://doi.org/10.1007/978-3-319-68270-9_1

[39] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *Journal of Automated Reasoning* 58, 1 (2017), 3–31. https://doi.org/10.1007/s10817-016-9388-y

[40] Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *9th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 1254)*, Orna Grumberg (Ed.). Springer, 72–83. https://doi.org/10.1007/3-540-63166-6_10

[41] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course (LNCS, Vol. 60)*, Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle (Eds.). Springer, 393–481. https://doi.org/10.1007/3-540-08755-9_9

[42] Volker Grimm, Eloy Revilla, Uta Berger, Florian Jeltsch, Wolf M. Mooij, Steven F. Railsback, Hans-Hermann Thulke, Jacob Weiner, Thorsten Wiegand, and Donald L. DeAngelis. 2005. Pattern-Oriented Modeling of Agent-Based Complex Systems: Lessons from Ecology. *Science* 310, 5750 (2005), 987–991. https://doi.org/10.1126/science.1116681

[43] Matthias Güdemann, Pascal Poizat, Gwen Salaün, and Lina Ye. 2016. VerChor: A Framework for the Design and Verification of Choreographies. *IEEE Transactions on Services Computing* 9, 4 (2016), 647–660. https://doi.org/10.1109/TSC.2015.2413401

[44] Henning Günther, Alfons Laarman, and Georg Weissenbacher. 2016. Vienna Verification Tool: IC3 for Parallel Software - (Competition Contribution). In *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 9636)*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, 954–957. https://doi.org/10.1007/978-3-662-49674-9_69

[45] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *27th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[46] Peter Haglich, Christopher A. Rouff, and Laura Pullum. 2010. Detecting Emergence in Social Networks. In *SocialCom/PASSAT*. IEEE Computer Society, 693–696.

[47] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software model checking for people who love automata. In *25th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[48] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*. William Kaufmann, 235–245.

[49] Francis Heylighen. 2016. Stigmergy as a Universal Coordination Mechanism I: Definition and Components. *Cognitive Systems Research* 38 (2016), 4–13. https://doi.org/10.1016/j.cogsys.2015.12.002

[50] C. A. R. Hoare. 1985. *Communicating sequential processes*. Prentice-Hall.

[51] Christopher-Eyk Hrabia, Marco Lützenberger, and Sahin Albayrak. 2018. Towards adaptive multi-robot systems: self-organization and self-adaptation. *Knowledge Engineering Review* 33 (2018), e16. https://doi.org/10.1017/S0269888918000176

[52] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Bounded Model Checking of Multi-Threaded C Programs via Lazy Sequentialization. In *26th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 585–602. https://doi.org/10.1007/978-3-319-08867-9_39

[53] Omar Inverso and Catia Trubiani. 2020. Parallel and Distributed Bounded Model Checking of Multi-Threaded Programs. In *25th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 202–216. https://doi.org/10.1145/3332466.3374529

[54] Adel Khaled and James Miller. 2017. Using π-calculus for Formal Modeling and Verification of WS-CDL Choreographies. *IEEE Transactions on Services Computing* 10, 2 (2017), 316–327. https://doi.org/10.1109/TSC.2015.2449850

[55] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[56] N. Koenig and A. Howard. 2004. Design and Use Paradigms for Gazebo, an Open-Source Multi-Robot Simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vol. 3. IEEE, 2149–2154 vol.3. https://doi.org/10.1109/IROS.2004.1389727

[57] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *26th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 17–34. https://doi.org/10.1007/978-3-319-08867-9_2

[58] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ Model Checking Made Symbolic. *PACMPL* 3, OOPSLA (2019), 123:1–123:30. https://doi.org/10.1145/3360549

[59] Igor V. Konnov, Helmut Veith, and Josef Widder. 2017. On the Completeness of Bounded Model Checking for Threshold-Based Distributed Algorithms: Reachability. *Information and Computation* 252 (2017), 95–109. https://doi.org/10.1016/j.ic.2016.03.006

[60] Savas Konur, Clare Dixon, and Michael Fisher. 2010. Formal Verification of Probabilistic Swarm Behaviours. In *7th International Conference on Swarm Intelligence (ANTS) (LNCS, Vol. 6234)*. Springer, 440–447. https://doi.org/10.1007/978-3-642-15461-4_42

[61] Panagiotis Kouvaros and Alessio Lomuscio. 2015. A Counter Abstraction Technique for the Verification of Robot Swarms. In *29th Conference on Artificial Intelligence (AAAI)*, Blai Bonet and Sven Koenig (Eds.). AAAI, 2081–2088.

[62] Johannes Lächele, Antonio Franchi, Heinrich H. Bülthoff, and Paolo Robuffo Giordano. 2012. SwarmSimX: Real-Time Simulation Environment for Multi-Robot Systems. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR) (LNCS, Vol. 7628)*, Itsuki Noda, Noriaki Ando, Davide Brugali, and James J. Kuffner (Eds.). Springer, 375–387. https://doi.org/10.1007/978-3-642-34327-8_34

[63] Akash Lal and Thomas W. Reps. 2008. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *20th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 5123)*, Aarti Gupta and Sharad Malik (Eds.). Springer, 37–51. https://doi.org/10.1007/978-3-540-70545-1_7

[64] Leslie Lamport. 2002. *Specifying Systems, the TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley.

[65] Frédéric Lang, Radu Mateescu, and Franco Mazzanti. 2019. Compositional Verification of Concurrent Systems by Combining Bisimulations. In *3rd World Congress on Formal Methods (FM) (LNCS, Vol. 11800)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer, 196–213. https://doi.org/10.1007/978-3-030-30942-8_13

[66] Axel Legay, Benoît Delahaye, and Saddek Bensalem. 2010. Statistical Model Checking: An Overview. In *International Conference on Runtime Verification (RV) (LNCS, Vol. 6418)*, Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Springer, 122–135. https://doi.org/10.1007/978-3-642-16612-9_11

[67] Alessio Lomuscio and Edoardo Pirovano. 2018. Verifying Emergence of Bounded Time Properties in Probabilistic Swarm Systems. In *27th International Joint Conference on Artificial Intelligence (IJCAI)*. ijcai.org, 403–409. https://doi.org/10.24963/ijcai.2018/56

[68] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. 2017. MCMAS: an open-source model checker for the verification of multi-agent systems. *Software Tools for Technology Transfer* 19, 1 (2017), 9–30. https://doi.org/10.1007/s10009-015-0378-x

[69] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Catalin Balan. 2005. MASON: A Multiagent Simulation Environment. *Simulation* 81, 7 (2005), 517–527. https://doi.org/10.1177/0037549705058073

[70] Maja J. Matarić. 1993. Designing Emergent Behaviors: From Local Interactions to Collective Intelligence. In *From Animals to Animats 2. Second International Conference on Adaptive Behavior*. MIT Press, 432–441.

[71] Robin Milner. 1975. Processes: A Mathematical Model of Computing Agents. In *Logic Colloquium '73*. Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 157–173. https://doi.org/10.1016/S0049-237X(08)71948-7

[72] Robin Milner. 1980. *A calculus of communicating systems*. LNCS, Vol. 92. Springer. https://doi.org/10.1007/3-540-10235-3

[73] Robin Milner. 1989. *Communication and concurrency.* Prentice Hall.

[74] Peter D. Mosses. 2004. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming* 60-61 (2004), 195–228. https://doi.org/10.1016/j.jlap.2004.03.008

[75] D.E Nadales Agut. 2012. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* PhD Thesis. Technische Universiteit Eindhoven. OCLC: 8087176745.

[76] Reza Olfati-Saber. 2006. Flocking for Multi-Agent Dynamic Systems: Algorithms and Theory. *IEEE Trans. Automat. Control* 51, 3 (2006), 401–420. https://doi.org/10.1109/TAC.2005.864190

[77] OMG. 2010. *Business Process Model And Notation (BPMN) - Version 2.0.* Technical Report dtc/2010-05-03.

[78] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chandra Krintz and Emery Berger (Eds.). ACM, 614–630. https://doi.org/10.1145/2908080.2908118

[79] H. Van Dyke Parunak. 1997. "Go to the Ant": Engineering Principles from Natural Multi-Agent Systems. *Annals of Operations Research* 75 (1997), 69–101. https://doi.org/10.1023/A:1018980001403

[80] Anna Philippou, Mauricio Toro, and Margarita Antonaki. 2013. Simulation and Verification in a Process Calculus for Spatially-Explicit Ecological Models. *Scientific Annals of Computer Science* 23, 1 (2013), 119–167. https://doi.org/10.7561/SACS.2013.1.119

[81] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. 2015. A Tuple Space for Data Sharing in Robot Swarms. In *9th EAI International Conference on Bio-Inspired Information and Communications Technologies (BICT)*. ICST/ACM, 287–294.

[82] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. 2012. ARGoS: A Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems. *Swarm Intelligence* 6, 4 (2012), 271–295. https://doi.org/10.1007/S11721-012-0072-5

[83] Gordon D. Plotkin. 1981. *A structural approach to operational semantics.* Technical Report DAIMI FN-19. Computer Science Department, Aarhus University. Reprinted in J. Log. Algebr. Program., 2004.

[84] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 46–57. https://doi.org/10.1109/SFCS.1977.32

[85] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-540-31980-1_7

[86] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. 2007. Towards the theoretical foundation of choreography. In *16th International Conference on World Wide Web (WWW)*, Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy (Eds.). ACM, 973–982. https://doi.org/10.1145/1242572.1242704

[87] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *26th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 8559)*. Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[88] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Ron K. Cytron and Peter Lee (Eds.). ACM, 49–61. https://doi.org/10.1145/199448.199462

[89] Craig W. Reynolds. 1987. Flocks, herds and schools: A distributed behavioral model. In *14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, Maureen C. Stone (Ed.), Vol. 21. ACM, 25–34. https://doi.org/10.1145/37402.37406

[90] Adrián Riesco. 2018. Model Checking Parameterized by the Semantics in Maude. In *14th International Symposium on Functional and Logic Programming (FLOPS) (LNCS, Vol. 10818)*, John P. Gallagher and Martin Sulzmann (Eds.). Springer, 198–213. https://doi.org/10.1007/978-3-319-90686-7_13

[91] Eric Rohmer, Surya P N Singh, and Marc Freese. 2013. V-REP: A Versatile and Scalable Robot Simulation Framework. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS*. IEEE, 1321–1326. https://doi.org/10.1109/IROS.2013.6696520

[92] Nima Roohi and Gwen Salaün. 2011. Realizability and Dynamic Reconfiguration of Chor Specifications. *Informatica* 35, 1 (2011), 39–49.

[93] Peter Schrammel and Daniel Kroening. 2016. 2LS for Program Analysis - (Competition Contribution). In *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 9636)*, Marsha Chechik and Jean"-"François Raskin (Eds.). Springer, 905–907. https://doi.org/10.1007/978-3-662-49674-9_56

[94] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. *PACMPL* 2, POPL (2018), 28:1–28:30. https://doi.org/10.1145/3158116

[95] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD) (LNCS, Vol. 1954)*, Warren A. Hunt Jr. and Steven D. Johnson (Eds.). Springer, 108–125. https://doi.org/10.1007/3-540-40922-X_8

[96] Andrei Stefanescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Rosu. 2016. Semantics-Based Program Verifiers for All Languages. In *SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 74–91. https://doi.org/10.1145/2983990.2984027

[97] D. J. Sumpter, G. B. Blanchard, and D. S. Broomhead. 2001. Ants and Agents: A Process Algebra Approach to Modelling Ant Colony Behaviour. *Bulletin of Mathematical Biology* 63, 5 (2001), 951–980. https://doi.org/10.1006/bulm.2001.0252

[98] Ichiro Suzuki and Masafumi Yamashita. 1999. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM J. Comput.* 28, 4 (1999), 1347–1363. https://doi.org/10.1137/S009753979628292X

[99] Leigh Tesfatsion. 2002. Agent-Based Computational Economics: Growing Economies From the Bottom Up. *Artificial Life* 8, 1 (2002), 55–82. https://doi.org/10.1162/106454602753694765

[100] Guy Theraulaz and Eric Bonabeau. 1999. A Brief History of Stigmergy. *Artificial Life* 5, 2 (1999), 97–116. https://doi.org/10.1162/106454699568700

[101] Emil Vassev, Roy Sterritt, Christopher A. Rouff, and Mike Hinchey. 2012. Swarm Technology at NASA: Building Resilient Systems. *IT Professional* 14, 2 (2012), 36–42.

[102] Germán Vidal. 2015. Symbolic Execution as a Basis for Termination Analysis. *Science of Computer Programming* 102 (2015), 142–157. https://doi.org/10.1016/j.scico.2015.01.007

[103] W3C. 2005. *Web Services Choreography Description Language - Version 1.0.* Technical Report.

[104] M. Winikoff. 2010. Assurance of Agent Systems: What Role Should Formal Verification Play? In *Specification and Verification of Multi-Agent Systems*. Springer. https://doi.org/10.1007/978-1-4419-6984-2_12

[105] Glynn Winskel. 1984. Synchronization Trees. *Theor. Comput. Sci.* 34 (1984), 33–82. https://doi.org/10.1016/0304-3975(84)90112-9

[106] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME) (LNCS, Vol. 1703)*, Laurence Pierre and Thomas Kropf (Eds.). Springer, 54–66. https://doi.org/10.1007/3-540-48153-2_6