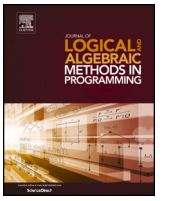
J. Log. Algebraic Methods Program. 144 (2025) 101048

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



A rewriting logic semantics for the analysis of P programs

Francisco Durán [a,*], Carlos Ramírez [b], Camilo Rocha [b], Nicolás Pozas [a]

[a] ITIS Software, Universidad de Málaga, Spain
[b] Departamento de Electrónica y Ciencias de la Computación, Pontificia Universidad Javeriana, Cali, Colombia

ARTICLE INFO

Keywords:
P
Maude
Model checking
Reachability analysis
Equational abstraction
LTL model checking
Statistical model checking

ABSTRACT

P is a domain-specific language designed for specifying asynchronous, event-driven systems. Its computational model is based on actors, i.e., on communicating state machines. This paper presents a formal semantics of P using rewriting logic, extending the language's verification capabilities. Implemented in Maude, a rewriting logic language, this semantics enables automated analysis of P programs, including reachability analysis, LTL model checking, and statistical model checking. Through illustrative examples, this paper demonstrates how this formalization significantly enhances P's verification capacities in practical scenarios.

1. Introduction

P is a domain-specific language meticulously crafted for specifying asynchronous event-driven systems, as referenced in [12, 46]. Its programs are essentially collections of state machines that rely on event-driven communication, reflecting the foundational model of communicating state machines (also known as actors) as described in the literature [29,1]. Notably, P facilitates automatic compilation of programs into executable code. These executables can then be paired with state exploration techniques, enabling the identification and rectification of design flaws.

This approach offers several well-established advantages. The model is highly adaptable, allowing for multiple recompilations as needed, with maintenance directly conducted on the state machine diagrams. Moreover, the properties verified on these diagrams are expected to hold true for the code automatically generated from them. P has already demonstrated its efficacy in implementing and verifying numerous industrial applications. Examples include the implementation and verification of the core of the USB device driver stack in Microsoft Windows 8, formal verification of core distributed protocols in Amazon S3's strong consistency launch, and formal specification and verification of the OTA protocol in AWS FreeRTOS.[1] For further examples and details, interested readers can refer to [46].

The design of the P language prioritizes the evaluation of program responsiveness, ensuring its ability to handle events promptly. To effectively model intricate distributed systems, the language offers flexibility by allowing specification of diverse machine types, states, actions, and events. The concept of *deferred* events in P allows specific events to be delayed when machines are in particular states. For testing purposes, programmers can also define the system's execution environment by creating nondeterministic *ghost* machines, which are removed during compilation. Additional features, such as monitors, hot states, and other elements, further support system specification and verification.

In practice, attempting to handle every event in each possible state often leads to a combinatorial explosion, making the formal verification of complex case studies challenging. To address this, P includes a dedicated tool for explicit-state bounded model checking

* Corresponding author.
E-mail address: fdm@uma.es (F. Durán).

[1] The web site of Amazon's real-time operating system for resource-constrained devices (FreeRTOS) is at https://aws.amazon.com/freertos.

https://doi.org/10.1016/j.jlamp.2025.101048
Received 11 March 2024; Received in revised form 29 January 2025; Accepted 11 February 2025

Available online 15 February 2025

alongside its compiler [12,4,11]. Recently, the P framework has been expanded to support statistical model checking of quantitative temporal logic formulas [18].

This work has a dual purpose. First, it establishes a formal semantics for the P language within rewriting logic. Second, it uses this semantics to make a suite of Maude tools available to P programs, such as a built-in LTL model checker [23], interactive theorem provers [10,45], and various other checkers [21,20]. Additional tools available for Maude specifications include model checkers for CTL, CTL*, and μ-calculus facilitated by connections to external tools [50]; a model checker tailored for strategy-controlled models [51]; and options for verifying quantitative properties through probabilistic and statistical model checking techniques [3, 55,52]. As showcased in this paper, the rewriting logic semantics for P enables exploration of various strategies, such as equational abstraction [40,9] and statistical model checking, addressing the combinatorial explosion challenges associated with P programs. The approach of employing rewriting logic as a semantic framework has been previously adopted for numerous other languages (see, for instance, [37,39,41]).

Given a rewriting logic semantics $\mathcal{R} = (\Sigma, E, R)$ for a language like P, the equational theory (Σ, E) establishes the state for any program within the language. This includes definitions for data types and deterministic non-observable transitions inherent to its semantics. The rewrite rules R serve as axioms defining the observable transitions within the system. In the context of P, these rules encapsulate the concurrent execution of multiple threads within an actors model framework and specify the semantics of nondeterministic choice operators like $ and choose (see [46]). Consequently, for reachability analysis and model checking, the transition system associated with any P program forms a directed graph, where vertices represent observable states and edges represent observable behaviors. Techniques such as equational abstraction can be applied by introducing additional equations to collapse states. Furthermore, Monte-Carlo simulations over the graph can be performed to statistically analyze quantitative properties of the system. These concepts are elucidated through a running example featuring a client-server distributed system.

The approach and tools introduced in this paper build upon the groundwork laid in [19] and complements the one reported in [18]. With respect to [19], this work completes a mathematically precise yet executable semantics for P in rewriting logic. With respect to [18], and as mentioned before, this unlocks novel verification capabilities for P such as LTL model checking. The former serves as a robust testbed for conducting rigorous experiments on language variants or new features. Meanwhile, the latter facilitates a harmonious integration of existing and novel testing and verification techniques under a unified semantic framework, introducing a fresh approach where modifications to P's compiler are unnecessary. In the long run, the formalization of P's semantics in Maude opens up the possibility of utilizing other tools from its formal environment. Specifically, as discussed in this paper, access to the statistical model checker introduced in [52] is immediately available. Other techniques and tools such as strategy-guided model checking [50], probabilistic model checking [52], and symbolic methods [15,30,33] are now accessible as well for future P program verification.

Significant effort has been invested to make the rewriting logic semantics $\mathcal{R}$ and statistical model checking accessible to P users with minimal effort and knowledge of Maude. Specifically, a compiler has been developed to transform P programs into the syntax of $\mathcal{R}$. This compiler has been tested with the testbed of programs publicly available in the latest release of P, and several other systems developed by the authors; for example, this parser successfully compiles the five examples in P's language tutorial. Two of these programs are used in this work to illustrate some of its capabilities. In addition to being fully managed by the rewriting logic semantics of P, they can be automatically verified, using methods such as statistical model checking. Several other examples have been developed by the authors, including the ones analyzed in Section 4.5. The rewriting logic semantics of P, the compiler that transforms P projects into Maude specifications, and several examples with instructions on how to use them are available from the public repository at https://github.com/PST-P/maude-p.

This paper is organized as follows. Section 2 overviews P and introduces a simple banking system example. Section 3 gives a quick overview of the Maude system and explains the key elements in the Maude specification of the P language. Section 4 summarizes the main types of model-checking analyses that are now available to P programs, illustrated on several examples. Section 5 compares the statistical model checking available thanks to this rewriting logic semantics and Maude's tool umaudemc, and proposal in using P's compiler and MultiVeStA. Section 6 concludes the paper.

2. An overview of P

P is a specialized programming language crafted for the development of asynchronous event-driven systems [12,46]. This approach involves programming systems through sets of interconnected state machines that exchange information via events. Typically, P programs are translated into executable C# code, enabling subsequent verification through state exploration methods to detect and rectify design flaws.

The foundational model of computation in P is rooted in the actors model [29,1], a well-established framework for concurrent computation. This model has garnered significant attention in recent decades due to its straightforward message-passing semantics, the rise of multi-core and cloud computing technologies, and the development of new programming languages built upon its principles, such as P and Scala. In the actors model, each actor is an independent concurrent entity that operates asynchronously and communicates with other actors through message passing. Every actor maintains a queue of messages awaiting processing, which are handled in the order of their arrival. Upon receiving a message, an actor can respond by making local decisions, spawning additional actors, or dispatching further messages. While actors possess their own private state, they can only affect each other indirectly through message passing. The execution of an actors model can be unbounded, continuing until all message queues become empty.

P's system implementation strategy offers a significant advantage by ensuring code consistency with the underlying model. This allows for effortless recompilation as necessary, facilitating maintenance directly on the state machine diagrams that represent the

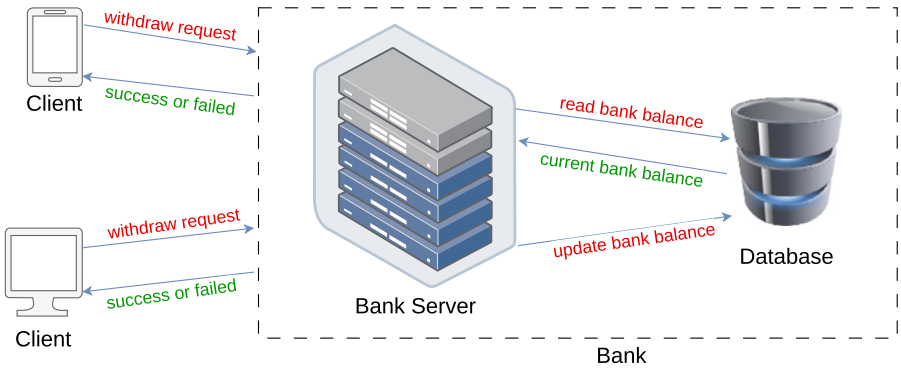


Fig. 1. The Client-Server running example.

actors model. The fundamental premise is that the properties verified on the state machine diagrams must also be upheld in the code generated automatically from the model.

The design of the P language is crafted to enable thorough evaluation of a program's responsiveness, ensuring its ability to promptly handle all events. However, owing to the need to model intricate distributed systems, the language exhibits complexity, offering provisions for specifying various machine types, states, actions, events, and more. For instance, to accommodate delayed processing of particular events within specific states, the language incorporates *deferred* events. Moreover, for testing purposes, programmers can simulate the system's execution environment by introducing nondeterministic *ghost* machines, which are subsequently removed during the compilation process. A program consists of multiple state machines that interact via events. Each machine encompasses a collection of states, actions, and local variables. Within these states and actions, code statements are defined to manipulate local variables, dispatch events to other machines, trigger local events, or invoke external C functions, primarily designated for data transfer operations. Upon receiving an event, a machine initiates transitions and executes actions, thereby executing the specified code fragments. Additionally, auxiliary functions can be employed for programming convenience.

In addition to its compiler, the language offers a tool for explicit-state bounded model checking. Various model checkers have been utilized in conjunction with different versions of P. Initially, an ad-hoc tool was created [12], followed by the adoption of Zing [4], and then Coyote [11] was integrated into its framework. More recently, P's framework has been extended to include statistical model checking of quantitative temporal logic formulas [18].

To exemplify some of the key features of P, one of the examples featured in the language tutorial [46] is used. Specifically, it is a banking system. Subsequently, the P code will be employed in the following sections to elucidate the Maude semantics of P and to demonstrate various analysis techniques applicable to P programs utilizing such semantics.

Fig. 1 illustrates the example system, portraying a conventional client-server application where clients engage with a bank to withdraw funds from their accounts. Alongside several clients, the system comprises a bank server and a backend database. Concurrently, multiple clients can send withdrawal requests to the bank, which in turn depends on the backend database to process these requests. The database serves the purpose of storing the account balance information for each client.

When a bank server receives a withdrawal request, it retrieves the client's account balance. If the request can be fulfilled based on the available balance, the server proceeds to execute the withdrawal, updates the account balance accordingly, and subsequently notifies the client with the updated balance. It is imperative that the balance of all accounts remains above or equal to $10. Therefore, any withdrawal request that would reduce the balance below this threshold must be rejected. The P code provided in the language's tutorial is designed to verify that, even in the presence of concurrent withdrawal requests from multiple clients, the bank consistently provides the accurate account balance to each client, and a withdrawal request is successful only if there are sufficient funds in the account.

The model consists of state machines for clients, bank servers and backend databases. Additional ghost machines represent the environment and the setting for the operation of specific configurations. Although P is a textual language, with the aim of improving the presentation, graphical representations of the different machines using standard statechart notation is used. Figs. 2-4 depict the Client, BankServer, and Database state machines, respectively.

The Client machine has a set of local variables used to store the local state of the state machine (upper-left corner). Upon creation, a Client machine gets a reference to its BankServer machine, which keeps an account with a unique identifier and a balance. Whilst the balance of her account is greater than 10, the client remains in the WithdrawMoney state, sending withdraw requests. Each request has an identifier, for which an index nextReqId is used. The amount to be withdraw is a random value in the range [1 … currentBalance]. This value is calculated in an auxiliary function WithdrawAmount() using the choose function.[2] The BankServer machine will respond indicating whether the operation was successful (resp.status == WITHDRAW_SUCCESS) or not. If successful, the client updates its balance. Note that several assert commands check the expected conditions.

[2] P provides three overloaded choose operations. The operation choose(), with no argument, nondeterministically returns true or false. choose(x), with x an integer value, returns a random value in the range [0, x). Finally, choose(c), with c a collection, returns a value from such a collection; if c is a map, it returns one of its keys.

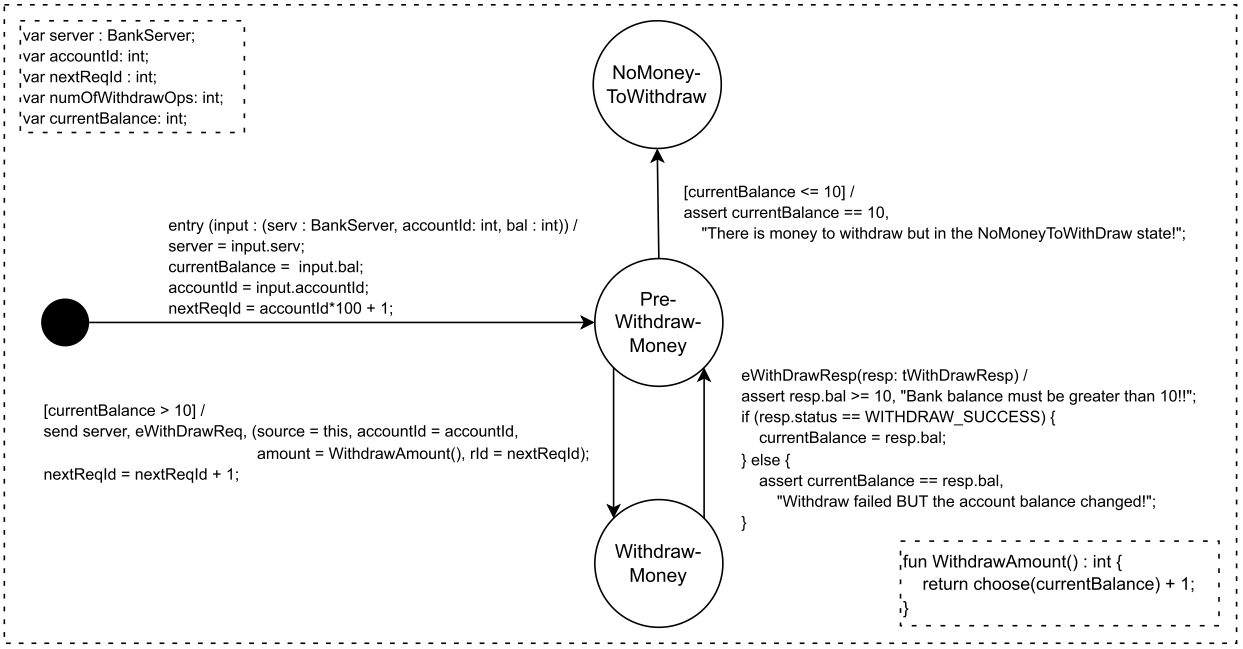


Fig. 2. The Client state machine.

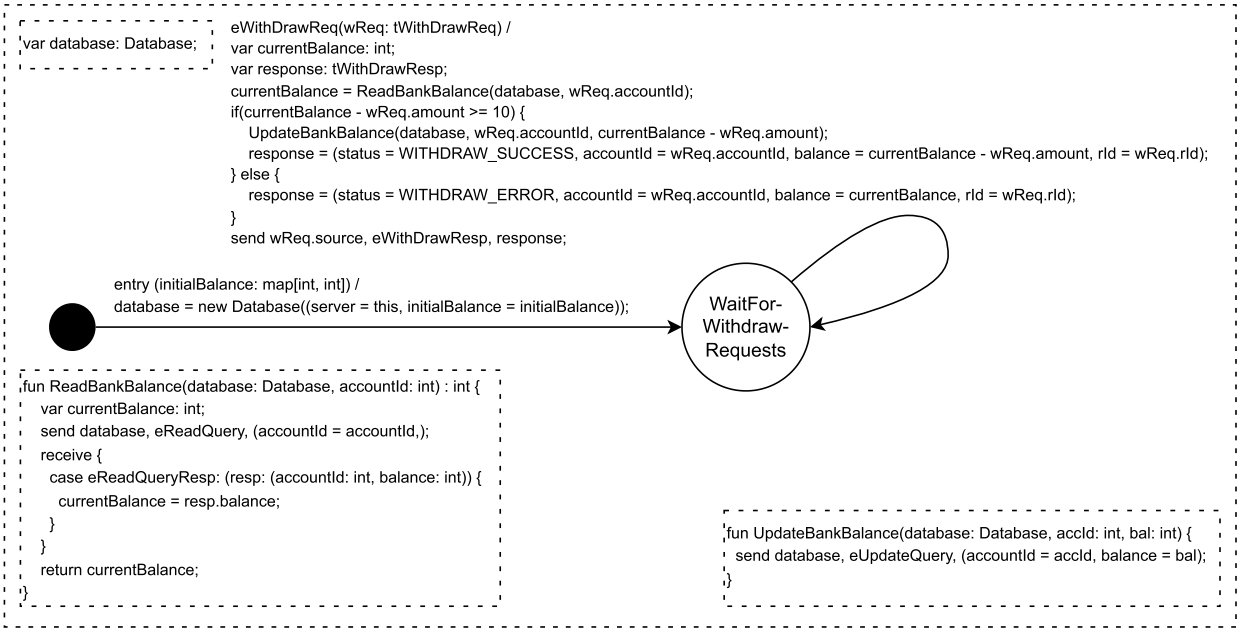


Fig. 3. The BankServer state machine.

Events eWithDrawReq and eWithDrawResp are used to communicate between the Client and the Server machines. Their payloads are, respectively, of declared types tWithDrawReq and tWithDrawReq.

```
event eWithDrawReq: tWithDrawReq;
event eWithDrawResp: tWithDrawResp;
type tWithDrawReq = (source: Client, accountId: int, amount: int, rId: int);
type tWithDrawResp = (status: tWithDrawRespStatus, accountId: int, balance: int, rId: int);
enum tWithDrawRespStatus { WITHDRAW_SUCCESS, WITHDRAW_ERROR }
```

The BankServer machine uses a Database machine as a service to store the bank balance for all its clients. On receiving a withdraw request (an event eWithDrawReq) from a client, it reads the current balance for the account. If there is enough money

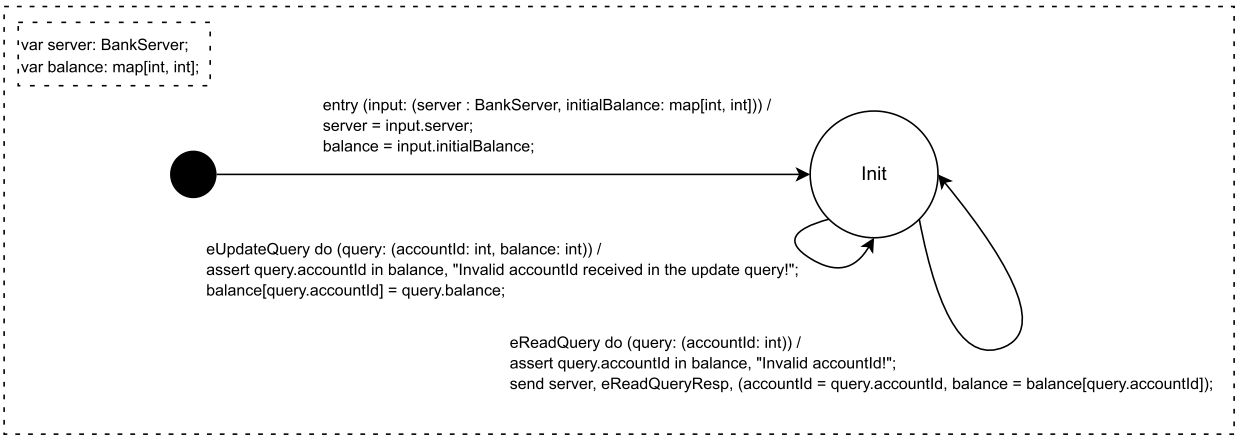


Fig. 4. The Database state machine.

in the account then it updates the new balance in the database after withdrawal and sends a response back to the client. Auxiliary functions ReadBankBalance(Database, int): int and UpdateBankBalance(Database, int, int) are in charge of reading the bank balance by sending a request to the database and waiting for its response, and updating the balance by sending the corresponding request to the database. The Database machine acts as a helper service for the bank server, storing the balance for each account.

The example is completed with test drivers. For instance, given functions CreateRandomInitialAccounts(int): map[int,int], which randomly initializes the balance for the specified number of client accounts, and SetupClientServerSystem(int), which sets up the client server system with one bank server and the number of clients specified as parameter, the machine TestWithMultipleClients defines a test driver that checks the system with a random number of clients in the range $[2, 4]$ (choose(3) + 2). Clients are created with an initial balance between 10 and 109 (choose(100) + 10). To get a better understanding of the nature of the code describing these machines, the code of the TestWithMultipleClients machine and its auxiliary functions is provided as found in [46] in Fig. 5.

```
machine TestWithMultipleClients {
  start state Init {
    entry {
      SetupClientServerSystem(choose(3) + 2);
    }
  }
}

fun CreateRandomInitialAccounts(numAccounts: int) : map[int, int] {
  var i: int;
  var bankBalance: map[int, int];
  while(i < numAccounts) {
    bankBalance[i] = choose(100) + 10;
    i = i + 1;
  }
  return bankBalance;
}

fun SetupClientServerSystem(numClients: int) {
  var i: int;
  var server: BankServer;
  var accountIds: seq[int];
  var initAccBalance: map[int, int];
  initAccBalance = CreateRandomInitialAccounts(numClients);
  server = new BankServer(initAccBalance);
  accountIds = keys(initAccBalance);
  while(i < sizeof(accountIds)) {
    new Client((serv = server, accountId = accountIds[i],
                balance = initAccBalance[accountIds[i]]));
    i = i + 1;
  }
}
```

Fig. 5. Code of the TestWithMultipleClients machine (from [46]).

The example in the tutorial includes the specification of safety and liveness properties. These properties are checked by another state machine, called monitor, which gets events from the different machines. In the P programming language, monitors are special constructs used to observe and enforce safety and liveness properties. They act as state machines that track specific properties of the system's execution. Monitors do not affect the execution of the system directly but raise alarms or log errors when certain properties are violated. This mechanism allows developers to specify and check correctness properties declaratively, ensuring that the system adheres to desired behaviors such as mutual exclusion, absence of deadlocks, or meeting response time requirements. Given the purpose of the proposed approach, monitor machines are not included here because the interest lies in verifying properties of P programs without restricting this task to a reduced number of traces.

This section concludes by emphasizing that only certain portions of the code have been discussed here. Figs. 2 through 4 provide visual representations of the machines, while the P programs themselves are expressed in textual code. Notably, the rewriting logic semantics of P, as elucidated in Section 3, takes an actual textual representation of a program as input. The entire code of the case study is publicly available at https://github.com/p-org/P/blob/master/Tutorial/1_ClientServer/ as part of the documentation of P.

3. A rewriting logic semantics for P

This section introduces the contributed rewriting logic semantics, denoted as $\mathcal{R}_{\mathsf{P}}$, for P. Initially, it provides an overview of rewriting logic along with its companion system and language, Maude. Subsequently, it elucidates the formal representation and execution of P programs facilitated by $\mathcal{R}_{\mathsf{P}}$. Crucial elements of the rewriting logic semantics are illustrated through the running example.

3.1. Rewriting logic and maude in a nutshell

Rewriting logic [38] serves as a semantic framework that consolidates various models of concurrency, offering a unified approach. This framework has found application in providing semantics for a diverse array of languages and formalisms. Notably, there exists a longstanding project on language semantics and the development of analysis tools for them using rewriting logic that has been extensively documented across multiple works (see, e.g., [41,43,42]).

Specifications in rewriting logic are called rewrite theories and they can be executed in Maude [9]. A rewrite logic theory is a tuple $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a membership equational logic [8] theory with Σ its signature, E a set of conditional equations, A a set of equational axioms (e.g., associativity, commutativity, and identity), so that rewriting is performed modulo A, and R is a set of labeled conditional rules.

In rewriting logic, a distributed system is axiomatized by an equational theory describing the set of states as an algebraic data type and a collection of conditional rewrite rules specifying the concurrent transitions. Rewrite rules are written as crl $[l]$: t => t' if C, with l a label, t and t' terms, and C a guard or condition. Rules describe the local, concurrent transitions that are possible in the system (i.e., when a part of the system state fits the pattern t, then it can be replaced by the corresponding instantiation of t'). The guard C acts as a blocking precondition in the sense that a conditional rule can only be fired if its condition is satisfied. Rules may be given without label or condition. Unlabeled and unconditional rules may be written as rl t => t'.

Conditions are either a Boolean expression or a conjunction of equalities u_i=v_i, sort membership axioms u_i:s_i, or matching equations of the form p_i:=u_i, where u_i and v_i are terms, p_i are pattern terms (irreducible terms with variables), and s_i are sorts. In its simplest form, pattern terms are just variables, with a functionality equivalent to *where* statements in typical functional programs.

3.2. Specification of P programs

Without diving into unnecessary detail, a machine definition includes its name (of type Name), its start state ([MaybeName]), a map associating a tuple to each of its states, including the payload of its initialization code, and another map associating the arguments and the code to be executed upon the reception of each event it can handle. Function declarations and the sets of events to be deferred and ignored complete the definition of a machine.

```
op [_,_,_,_,_,_,_,_] :
   Name
   [Maybe{Name}]
   Map{Name,
       Tuple5{NamedTypeSequence,
              Maybe{Code},
              Map{EventName, Tuple2{NamedTypeSequence, Code}},
              Set{EventName},
              Set{EventName}}}
   Set{Function}
   Set{EventName}
   Set{EventName}
   Memory -> Machine .
```

A machine instance is modeled as an actor, with a unique identifier (of type MachId), a type (Name), and an unbounded queue of incoming events (Queue{Event}). Under execution, the representation of a machine also keeps its current state (Maybe{Name}), the code currently being executed ([Maybe{Code}]), and its memory stack (MemoryStack). The state of a machine under execution

is completed with information on ignored and deferred events, including a second events queue with those waiting to be handled, and local function definitions. As seen, for example, for the Client and BankServer machines, function declarations local to machines have access to their state. These structures are represented as a term of sort MachExecState using the following constructor:

```
op [_,_,_,_,_,_,_,_,_,_] : MachId Name Maybe{Name} Queue{Event} Queue{Event}
      [Maybe{Code}] Set{Function} Set{EventName} Set{EventName} MemoryStack
        -> MachExecState
```

A system is then represented as a term of sort SystemExecState, composed of a set of executing machines (ExecState), indexes for the generation of fresh names and pseudorandom numbers, sets of declarations of machines, global functions, and tests (System), and a string to collect the log of the execution.

```
op [_,_,_,_,_,_] : ExecState Nat Nat System String -> SystemExecState .
op ((_,_,_)) : Set{Machine} Set{Function} Set{TestDecl} -> System .
```

3.3. Supporting the syntax of P programs

Appropriate operation declarations define the syntax of the P language. In the signature of $\mathcal{R}_{\mathsf{P}}$ there are sorts like BoolExpr, VarDecl or Sentence for the different elements of the language, and operators declarations like the ones below that will allow us to represent each of the different elements in a P program.

```
sorts VarDecl VarDecl* Sentence Code CodeStack .
subsort VarDecl < Sentence VarDecl* < Code < CodeStack .
op skip : -> VarDecl* .
op __ : VarDecl* VarDecl* -> VarDecl* [assoc id: skip prec 50] .
op __ : Code Code -> Code [assoc id: skip prec 50] .
op noCode : -> CodeStack .
op _<<|_ : CodeStack CodeStack -> CodeStack [assoc id: noCode prec 53] .

op while (_) {_} : BoolExpr Code -> Sentence .
op send_,_,_; : MachExpr EventName Expr -> Sentence .
op print_; : Expr -> Sentence .
op var_:_; : VarId TypeName -> VarDecl .
...
```

For example, the following declarations are generated as part of the Maude representation of the Client-Server model to represent the above machine TestWithMultipleClients and the function CreateRandomInitialAccounts:

```
op TestWithMultipleClients# : -> MachineDecl .
ops TestWithMultipleClients Init : -> Name .
op TestWithMultipleClients : -> UserDefinedTypeName .
eq TestWithMultipleClients#
  = machine TestWithMultipleClients {
      start state Init {
        entry {
          SetupClientServerSystem(choose(3) + 2) ;
        }
      }
    } .
```

```
op CreateRandomInitialAccounts# : -> FunDecl .
op CreateRandomInitialAccounts : -> Name .
ops numAccounts i : -> IntVarId .
op bankBalance : -> MapVarId .
eq CreateRandomInitialAccounts#
  = fun CreateRandomInitialAccounts(numAccounts : int) : map[int, int] {
      var i : int ;
      var bankBalance : map[int, int] ;
      while(i < numAccounts) {
        bankBalance[i] = choose(100) + 10 ;
        i = i + 1 ;
      }
      return bankBalance ;
    } .
```

Please, compare this code with the one in Fig. 5. For each machine and function definition X, a constant X# of type MachineDecl and FunDecl, respectively, is added. Constants for state names (Init) and variables (i, numAccounts) are also declared of appropriate types.

With corresponding declarations for all user-defined types, events, machines, functions, and tests, a constant init of type System is generated to represent the system, incorporating all the necessary elements.

```
op init : –> System .
eq init
  = init([CreateRandomInitialAccounts#] [SetupClientServerSystem#]
         [tcMultipleClients#] [tcTwoClients#] [tcSingleClient#]
         [readBankBalance#] [updateBankBalance#]
         [TestWithMultipleClients#] [TestWithTwoClients#] [TestWithSingleClient#]
         [Client#] [BankServer#] [Database#]) .
```

3.4. Execution of P programs

Given the appropriate declarations defining the structures of P programs as a membership-equational theory (Σ, E), the rewriting logic semantics $\mathcal{R}_{\mathsf{P}} = (\Sigma, E, R)$ is given as a set of transformation rules R. Specifically, each of the instructions that may appear in the code of a P program is handled by one or several rules. Just to get a taste of the kind of rules that the formalization of P includes, consider the cases of assignment and event sending. These are two key operations, which illustrate the handling of memory and the exchange of events between machines.

The following two equations handle the case of an assignment of an expression to a variable of a basic type (int, float, string or bool):

```
eq [assignment] :
   [[MId, Id, St, Events, VId = E ; Code <<| CS, FS, MS] ES, S, Sys, Log]
 = [[MId, Id, St, Events, eval(E) <| VId = @ ; Code <<| CS, FS, MS] ES, S, Sys, Log] .
eq [assignment] :
   [[MId, Id, St, Events, result(E) <| VId = @ ; Code <<| CS, FS, MS] ES, S, Sys, Log]
 = [[MId, Id, St, Events, Code <<| CS, FS, setV(VId, E, MS)] ES, S, Sys, Log] .
```

If the first sentence at the top of the execution stack of one of the machines' execution states is a sentence of the form VId = E;, representing the assignment of the expression E to the variable VId, then, first, the expression is evaluated in a new layer of the execution stack. Once evaluated, the second equation takes the result of the evaluation result(E) and updates the value associated with the variable VId in the memory stack of the machine (MS) using the auxiliary function setV.

The following statements handle the instruction that sends an event EN without payload to a target machine ME:

```
eq [event–send] :
   [[MId, Id, St, Events, send ME, EN ; Code <<| CS, FS, MS] ES, S, Sys, Log]
 = [[MId, Id, St, Events, eval(ME) <| send @, EN ; Code <<| CS, FS, MS] ES, S, Sys, Log] .
rl [event–send1] :
   [[MId, Id, St, Events, result(MId') <| send @, EN ; Code <<| CS, FS, MS]
    [MId', Id', St', Events', CS', FS', MS'] ES, S, Sys, Log]
=> [[MId, Id, St, Events, Code <<| CS, FS, MS]
    [MId', Id', St', Events' << EN{noValue}, CS', FS', MS'] ES, S, Sys, Log]
```

As before, the first equation evaluates the machine expression ME. The rule manages the event sending by placing it in the event queue of the target machine. In the case of events with arguments, as for function invocations or new instance creations, each of the arguments must be evaluated before handling the actual sending, invocation or creation, but the procedure is basically the same one. Note also that in the same way there is a code stack, there is a corresponding memory stack, allowing the correct handling of nested invocations.

Note that some of the transitions above are written as equations and others as rules. Following Farzan and Meseguer [26], some of the rules are turned into equations to reduce the state space explosion.

The theory $\mathcal{R}_{\mathsf{P}}$ is itself an interpreter for P programs. Therefore, it allows the execution of P programs and the use of the tools in the Maude formal environment, to carry on reachability analysis, model checking, and statistical model checking. To execute the Client-Server example, the frewrite command can be utilized, as demonstrated in Fig. 6. Please note the ellipses and additional new lines inserted for improved readability. Note also that the command was executed with a limit on the number of rewrites. The program is non-terminating and, without such a bound, it would continue executing indefinitely.

The output in Fig. 6 is not relevant by itself; however, it helps to observe the system structure and gather ideas on how to query systems to get useful information. Indeed, the code in the machines has been replaced with ellipses. Definitions of functions and other non-relevant details for the current purpose of illustrating the use of the semantics, including information on which machines are alive, in what states they are, and what are the values of each of their variables, have been also intentionally neglected. The output includes relevant information. Specifically, three clients have been created. The first one has identifier id(7), account identifier 2, initial balance 58, and current balance 11. The second client has identifier id(8), account identifier 1, and initial and current balance 103. The third one has identifier id(9) and account identifier 0, and initial and current balance 24. Recall that the initial values are given in the CreateRandomInitialAccounts function above, which gives nondeterministic values between 10 and 109. As the final log shows, some withdraws have been attempted, some have been successful and others have not. Also note that the amount to withdraw is a value between 1 and currentBalance – 1 (choose(currentBalance) + 1), but withdraw operations are successful only if the remaining balance is over 10 after the operation. There is at least one more relevant piece of information in the output, in this case something that is not shown: all asserts have been timely checked, and since no error message has been reported and the execution proceeded normally, it may be concluded that all checks were successful for this specific execution.

Interesting questions about this program regarding, for instance, multiple execution paths and temporal properties can be answered with the help of $\mathcal{R}_{\mathsf{P}}$, as will be seen in Section 4.

```
Maude> frewrite [100] execute(tcMultipleClients, init) .
rewrites: 4664 in 45ms cpu (46ms real) (103183 rewrites/second)
result SystemExecState:
 [ [ id(0), TestWithMultipleClients, Init, ..., ..., ..., mtMemory <m| [this –> id(0)] ]
   [ id(5), BankServer, WaitForWithdrawRequests, ..., ..., ...,
     [accountId –> 2] [database –> id(6)] [currentBalance –> 0]
     <m| [currentBalance –> 0] [response –> noValue]
         [wReq –> (source = id(7), accountId = 2, amount = 5, rId = 210)]
     <m| [this –> id(5)] [database –> id(6)]
         [initialBalance –> map{0 |–> 24, 1 |–> 103, 2 |–> 58}] ]
   [ id(6), Database, Init, ..., ..., ...,
     [query –> (accountId = 2)]
     <m| [this –> id(6)] [server –> id(5)]
         [balance –> map{0 |–> 24, 1 |–> 103, 2 |–> 11}]
         [input –> (server = id(5),
                    initialBalance = map{0 |–> 24, 1 |–> 103, 2 |–> 58})] ]
   [ id(7), Client, WithdrawMoney, ..., ..., ...,
     [index –> 0] <m| [this –> id(7)] [accountId –> 2] [currentBalance –> 11]
     [server –> id(5)] [nextReqId –> 211] [numOfWithdrawOps –> noValue]
     [input –> (serv = id(5), accountId = 2, bal = 58)] ]
   [ id(8), Client, WithdrawMoney, ..., ..., ...,
     [index –> 0] <m| [this –> id(8)] [accountId –> 1] [currentBalance –> 103]
     [server –> id(5)] [nextReqId –> 101] [numOfWithdrawOps –> noValue]
     [input –> (serv = id(5), accountId = 1, bal = 103)] ]
   [ id(9), Client, WithdrawMoney, ..., ..., ...,
     [index –> 0] <m| [this –> id(9)] [accountId –> 0] [currentBalance –> 24]
     [server –> id(5)] [nextReqId –> 1] [numOfWithdrawOps –> noValue]
     [input –> (serv = id(5), accountId = 0, bal = 24)] ],
   22,
   (..., ..., ...),
   "Withdrawal with rId = 201 failed, account balance = 58\n
    Still have account balance = 58, lets try and withdraw more\n
    Withdrawal with rId = 202 succeeded, new account balance = 57\n
    Still have account balance = 57, lets try and withdraw more\n
    ..." ]
```

Fig. 6. Execution of the tcMultipleClients test case.

3.5. Compilation of P programs

P's own compiler translates the code into an intermediate representation (IR). This IR is a lower-level representation of the program that abstracts away the high-level constructs while preserving the logical flow and behavior of the original P code. From the IR, the compiler generates C code, which is executable and retains the logic and state machine behavior defined in the original P program.

This paper presents a custom parser that translates P programs into the syntax of the rewriting logic semantics of the language, contrasting with the approach in [18], which modified P's own compiler. The parser is written in Python and utilizes the Lark parsing toolkit (https://github.com/lark-parser/lark). With this toolkit, the P grammar is specified to produce an abstract syntax tree (AST) for the original P program. While P programs can be written in Maude in a form nearly accepted by the P compiler, processing the AST is necessary to generate the Maude modules corresponding to the P semantics.

During the processing of the AST, the parser identifies the names of the declared types, events, functions, machines, monitors and modules and consistently generate the Maude code for the declaration of some constants associated with these elements. This task also involves the replacement of the names in the P program that are used in several parts of the code with different datatype being as each datatype is mapped to a different Maude sort. For example, a variable x, which is declared and used as an integer in some function, could be also declared as a float in some payload function. Then, in the generated Maude file the integer occurrence of the variable x will be mapped to the constant x$Int of the sort IntVarId whereas the float occurrence will be associated with the constant x$Float of the sort FloatVarId.

Additionally, the parser will translate some blocks of P code with some additional spaces between tokens to help Maude's parser. Maude identifiers can contain any characters, and only parentheses and commas break tokens. For example, something like 2+3 is a valid identifier in Maude. The custom parser tokenizes it as 2␣+␣3.

In this way, from the files defining any P program, the corresponding Maude files with the required structure are thus generated.

4. Verification of P programs

This section delves into various forms of analysis accessible to $\mathcal{R}_{\mathsf{P}}$. These encompass state-space exploration, and LTL and statistical model checking. Additionally, it introduces an equational abstraction that enables model checking with $\mathcal{R}_{\mathsf{P}}$. These types of analyses are exemplified using the Client-Server program.

4.1. State-space exploration

State-space exploration can be employed, for instance, by varying the number of clients in each execution, assigning different initial balances to their accounts, changing the amount of money to withdraw, or simply delivering events in one order or another. Each possible behavior gives place to an alternative potential path that needs to be analyzed. In essence, the objective is to traverse all feasible paths at any conceivable depth. These can be achieved by using Maude's search command.

Starting at some particular initial state, state-exploration with $\mathcal{R}_{\mathsf{P}}$ can be used to find a state satisfying a given condition, or the satisfaction of some given invariant. However, the state explosion may make it impractical. This is what happens in the case of the running example. A search bound can be set, but even with a small depth, the number of states is huge. The test case is creating between 2 and 4 clients, each with an initial balance in the range [10..110), and then each withdraw operation will be tried for an amount between one and the current balance of the account. In addition to all this variability, the code being executed in each machine gets modified with the execution of each sentence, requests are numbered and get increased (nextReqId), text is added to the log, ... A perfect combination for a blast!

A few things can be done to verify the program before losing hope. First, consider a simpler test case. The tcTwoClients test case creates two clients, a bank, and a database, and sets an initial balance of 20 for each of the clients. Bounded search can be executed for states in which the balance of a client has gone under 10 with the following command.

```
Maude> search [1, 15] in CLIENT−SERVER :
          execute(tcTwoClients, init)
          =>*
          [ ES:ExecState
            [ Id1:MachId, Client,
              State:Name, Q:Queue{Event}, C:Code, FDS:Set{Function},
              MS:MemoryStack <m| M:Memory [currentBalance −> N:Nat] ],
            N2:Nat, S:System, L:String ]
          such that N:Nat < 10 = true .

No solution.
states: 318831 rewrites: 19409934 in 409859ms cpu (252752ms real) (47357 rews/sec)
```

In this command, the initial state execute(tcTwoClients, init) is specified, which initiates the tcTwoClients test case on a system with all the required machines, functions, and test cases loaded (init). As expected, the response is positive, because there are no reachable states up to the fixed maximum depth in which a client has a balance below the established limit. However, even though the maximum depth is set to 15, the exploration took around 6.8 minutes.

4.2. An equational abstraction of P programs

To streamline the size of a state-space exploration task, equational abstractions can be used [40]. If the abstraction renders the state space finite, the search command can be employed to explore all potential behaviors without imposing a fixed depth limit. By constraining the search to a specified depth, confidence in the property's validity or the impossibility of reaching a certain scenario is bolstered. However, it is important to acknowledge that this approach is inherently incomplete, as there may always exist a counterexample at a greater depth. One viable strategy entails verifying properties not on the original infinite-state system, but rather on an abstraction thereof.

An abstraction of the system can effectively diminish the size of the state space by furnishing a quotient of the original system. If the reduction not only shrinks the state space but also renders it finite, then the procedure becomes complete, allowing exploration of the system to any depth. Moreover, if the reachable state space becomes finite through the abstraction, model checking can be applied to it. In the realm of model checking, an abstraction simplifies the task of determining whether an infinite-state system satisfies a temporal logic property by transforming it into the evaluation of that property on a finite state abstraction of the system.

Equational abstraction [40] is a simple, but powerful, method for defining abstractions. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, the method consists on adding equations to the theory, that is, in defining an abstract theory as a rewrite theory $\mathcal{A} = (\Sigma, E \cup E', R)$, with E' a set of equations that collapses the infinite set of reachable states into a finite set. As will be seen in Section 4.3, this method can also be used to verify LTL formulas [40].

The module CLIENT-SERVER-ABSTRACTION in Fig. 7 defines an abstraction of the theory defined by the CLIENT-SERVER module. Specifically, it provides three equations. The first two equations deal with the reasons that make the system infinite. The third one drastically reduces the size of the state space.

The Client-Server program prints messages showing how things are going, whether the withdraw requests are successful or not, or whether an assertion fails. These messages are of course important for the user, but are irrelevant for the checking of invariants. The equation abstraction-log removes any text added to the log. Note that the equation is conditional, it will be applied only if there is some text in it.

The equation abstraction-req-id assigns a zero to the variable nextReqId of clients. Each Client machine has one of these nextReqId variables, which keeps increasing. In fact, this variable is wrongly used, since it is initialized using the expression accountId * 100 + 1, supposedly to keep them unique. But obviously this only works if there are no more than one hundred transactions per client. The fact is that they are used only locally, and that they are not used at all. Since no request is sent until after the previous request has been appropriately acknowledged, this variable can be eliminated.

```
mod CLIENT-SERVER-ABSTRACTION is
  generated-by CLIENT-SERVER .

  var MId CId DId : MachId .
  var Events CEQ DEQ : Queue{Event} .
  var CS CCS DCS : CodeStack .
  var FS CFS DFS : Set{Function} .
  var MS CMS DMS : MemoryStack .
  var MM CMM DMM : Memory .
  var ES : ExecState .
  var S N V V' : Nat .
  var Sys : System .
  var Log : String .
  var M : Map{UniversalType,UniversalType} .
  var X CSt DSt : Name .

 ceq [abstraction-log] : [ES, S, Sys, Log] = [ES, S, Sys, ""] if Log =/= "" .
  eq [abstraction-req-id] :
     [MId, Client, X, Events, CS, FS, MS <m| MM [nextReqId -> s N]]
   = [MId, Client, X, Events, CS, FS, MS <m| MM [nextReqId -> 0]] .
 ceq [abstraction-balance] :
       [ [ CId, Client, CSt, CEQ, CCS, CFS,
           CMS <m| [accountId -> S] [currentBalance -> N] CMM ]
         [ DId, Database, DSt, DEQ, DCS, DFS,
           DMS <m| [balance -> map{S |-> N, M}] DMM ]
         ES:ExecState, V, Sys, Log ]
    = [ [ CId, Client, CSt, CEQ, CCS, CFS,
           CMS <m| [accountId -> S] [currentBalance -> 12] CMM ]
         [ DId, Database, DSt, DEQ, DCS, DFS,
           DMS <m| [balance -> map{S |-> 12, M}] DMM ]
         ES:ExecState, V, Sys, Log ]
    if N > 12 .
endm
```

Fig. 7. The CLIENT-SERVER-ABSTRACTION module.

Whilst the previous two equations make the state space finite, the third equation, abstraction-balance, significantly reduces its size. As pointed out in the previous section, one of the main reasons for such a huge number of states is the possible values for the balances of clients, and associated with balance values, the range of possible amounts for withdraw operations. However, from an analysis point of view, this only needs to be known for amounts over 10: withdraw operations can be of valid or invalid amounts. That is, there is no need to distinguish between balances 15, 20, or 25, and given a certain balance, whether the withdraw is for 1, 5, or 10. It would be enough to have certain amount X and withdraw requests that would leave the balance over 10, at 10, and below 10. The equation abstraction-balance leaves the balance of an account with value 12, so that potential withdraw request will be attempted with values 1, leaving the balance with 11; 2, leaving it with 10; or a value in the range [3..12], which should fail. Please, note that the change takes place simultaneously at the client and at the database, making sure the information is consistent.

Although more powerful abstractions can be thought of, this one clearly exemplifies the technique, and still gets a significant size reduction. Given this abstraction, a new search for states with clients with balances under 10 can be executed.

```
Maude> search in CLIENT-SERVER-ABSTRACTION :
            execute(tcTwoClients, init)
            =>* [ ES:ExecState
                  [ Id1:MachId,
                    Client,
                    State:Name,
                    Q:Queue{Event},
                    C:Code,
                    FDS:Set{Function},
                    MS:MemoryStack <m| M:Memory [currentBalance -> N:Nat]],
                N2:Nat, S:System, L:String ]
                such that N:Nat < 10 .

No solution.
states: 66507 rewrites: 9941579 in 107289ms cpu (80140ms real) (92660 rewrites/second)
```

The execution time has been significantly reduced to less than 2 minutes, but, more importantly, note that no limit has been given to the command. Since the state space has been made finite, the search command can be executed without bound, serving as an actual proof.

Note also that the number of states has been significantly reduced. With bound 15 the search command in Section 4.1 explored 318831 states in 409 seconds. The above exhaustive exploration has visited 66507 states in 107 seconds.

```
mod CLIENT−SERVER−PREDS is
  inc CLIENT−SERVER .
  inc SATISFACTION .

  subsort SystemExecState < State .

  op machine_at state_ : Name Name −> Prop .
  op event_for_ : EventName Name −> Prop .
  op all clients at state_ : Name −> Prop .

  var MId : MachId .
  vars Events Events' : Queue{Event} .
  var CS : [CodeStack] .
  var FS : Set{Function} .
  vars MS MS' : MemoryStack .
  var ES : ExecState .
  var S : Nat .
  var Sys : System .
  var Log : String .
  vars M X Y : Name .
  var Expr : [Expr] .
  var EN : EventName .

  eq [ [ MId, M, X, Events, CS, FS, MS ] ES, S, Sys, Log ]
     |= machine M at state X = true .
  eq [ [ MId, M, X, Events << EN{Expr} << Events', CS, FS, MS ] ES, S, Sys, Log ]
     |= event EN for M = true .
  eq [ [ id(0), M, Init, mtq, skip, empty, MS ]
       [ MId, Client, X, Events, CS, FS, MS' ] ES, S, Sys, Log ]
       |= all clients at state Y
    = X .==. Y
      and
      [ [ id(0), M, Init, mtq, skip, empty, MS ] ES, S, Sys, Log ]
        |= all clients at state Y .
  eq [ [ id(0), M, Init, mtq, skip, empty, MS ] ES, S, Sys, Log ]
       |= all clients at state X
    = true [owise] .
endm
```

Fig. 8. The CLIENT-SERVER-PREDS module.

4.3. LTL model checking

Any admissible rewrite theory can be model checked against any linear temporal logic (LTL) formula if the set of states reachable from a given initial state is finite. With the abstraction given before, the state space reachable from the given initial state is finite. Therefore, Maude's model checker for LTL [23,9] can be used as a decision procedure for satisfaction of LTL properties.

Several concepts must be introduced before verifying any formula using Maude's LTL model checker. First, given a set AP of *atomic propositions*, formulas of the *propositional linear temporal logic* LTL(AP) are defined inductively as usual, with operators `True` ($\top$), False ($\bot$), atomic propositions (defined by the user), conjunction ($\wedge$), implication ($\rightarrow$), eventually ($\Diamond$), henceforth ($\Box$), etc.

Kripke structures can be associated with rewrite theories. Essentially, a Kripke structure is a total transition system that has unary state predicates associated with each of its states. Specifically, a *Kripke structure* is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ such that A is a set of states, $\rightarrow_{\mathcal{A}}$ is the transition relation (a total binary relation), and $L : A \longrightarrow \mathcal{P}(AP)$ is the labeling function, which associates to each state $a \in A$ the set $L(a)$ of the atomic propositions in AP that hold in the state a.

The semantics of LTL(AP) is defined by means of a satisfaction relation $\mathcal{A}, a \models \varphi$ between a Kripke structure $\mathcal{A}$ having AP as its atomic propositions, a state $a \in A$, and an LTL formula $\varphi \in$ LTL(AP). What needs to be done to associate a Kripke structure to a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ then is to specify the sort of the states in the signature Σ, and the atomic propositions. To do this, Maude provides a sort State, which will be declared as supersort of the sort of the states of the system, SystemExecState in the task at hand. The semantics of the state predicates will be defined using the predefined operator

```
op _|=_ : State Prop ~> Bool [frozen] .
```

Specifically, each state predicate will be declared as an operator of sort Prop, and its semantics will be defined by equations.

The module CLIENT-SERVER-PREDS in Fig. 8 includes the necessary definitions. First, observe the subsort declaration SystemExecState < State, making terms of sort SystemExecState the states on which the model checker will operate. The use of the model checker on the P program will be illustrated below with properties such as it is always the case that eventually all clients will be in the NoMoneyToWithDraw state, or that all event eWithDrawReq will eventually produce a eWithDrawResp event. With this goal in mind, state predicates indicating whether a machine is in a given state (machine_at state_), whether there is a certain event in the queue of a given machine (event_for_), or whether all clients are in a certain state (all clients at state_) are required.

The equations defining the satisfaction of the atomic propositions only need to specify the positive cases. For the first two propositions (event_for_ and machine_at state_), the equations just match the corresponding situations. A state satisfies the machine M at state X proposition if there is a machine M in state X. Similarly, a state satisfies the event EN for M proposition if there is a machine M that has an event EN in its event queue. The equations for the third proposition illustrate a recursive definition of the satisfaction of propositions. To check whether all the Client machines are in some specific state, it is key to check whether the state of each of the machine matches the given one. Note that the test case machine, with identifier id(0), is included in the equations with empty code (skip), making sure that the check is not performed before all the machines in the case have been created. Since the check can operate on systems with any number of clients, it will also be trivially true if there are no client machines, or if the check is done before they are created.

Before using the model checker, a module importing all the above definitions and the model checker module needs to be specified.

```
mod CLIENT-SERVER-CHECK is
  inc CLIENT-SERVER-ABSTRACTION .
  inc CLIENT-SERVER-PREDS .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .
endm
```

The use of the model checker is illustrated with the verification of two LTL formulas. The first one states that some time after a BankServer machine receives an eWithDrawReq event, a Client machine will receive an eWithDrawResp event — note that the formula does not check that the client making the request is the one receiving the response.

```
Maude> red modelCheck(
         execute(tcTwoClients, init),
         [] (event eWithDrawReq for BankServer
             -> <> event eWithDrawResp for Client)) .
rewrites: 7516427 in 82866ms cpu (76783ms real) (90705 rewrites/second)
result Bool: true
```

The response is positive.

The second checked formula specifies that it is always true that eventually all clients will be in state NoMoneyToWithDraw. In this case, the response is negative, and the model checker gives a lengthy counterexample with an infinite execution path that demonstrates that it is not always the case. The path has been replaced with an ellipsis, but it shows a path in which an infinite series of failing withdraw requests are submitted to the bank.

```
Maude> red modelCheck(
         execute(tcTwoClients, init),
         [] <> all clients at state NoMoneyToWithDraw) .
rewrites: 4062 in 35ms cpu (35ms real) (115005 rewrites/second)
result ModelCheckResult: counterexample(...)
```

4.4. Statistical model checking of P programs

A different approach to tackle the inherent issue of state explosion in analyzing $\mathcal{R}_{\mathsf{P}}$ programs involves utilizing statistical model checking [32]. This method merges simulation and statistical techniques to examine timed and probabilistic systems. It offers a robust and scalable solution, especially when contrasted with numerical probabilistic model checking methods that face challenges with expanding state spaces. The umaudemc tool [50] is employed to illustrate its use on the running example by using the scheck command to perform Monte Carlo simulations on the rewrite theory $\mathcal{R}_{\mathsf{P}}$ augmented with probabilities. This procedure facilitates the estimation of a quantitative temporal expression expressed in the QuaTEx query language [2].

The umaudemc tool offers a broad array of quantitative techniques for Maude specifications. It includes both internally implemented methods and interfaces with external tools. Specifically, it may employ PRISM [31] and Storm [28] as backend tools for tasks such as computing transient and steady-state probabilities, checking and evaluating probabilities of LTL, PCTL*, and TCTL formulas, and determining expected values of arbitrary functions on the states. Additionally, umaudemc directly handles the estimation of quantitative temporal queries through Monte Carlo simulations, while also granting direct access to MultiVeStA [55].

While the running example offers the potential for compelling probabilistic and statistical analysis, let us provide only a brief overview of the capabilities by illustrating a simple example of statistical model checking. Both the umaudemc tool and MultiVeStA utilize the Quantitative Temporal Expressions (QuaTex) language. QuaTex was initially introduced in [2] and has also been employed in PVeStA [3].

The core module of umaudemc's statistical model checker calculates quantitative temporal expressions through multiple runs. Each simulation produces a value for every evaluation statement. The simulation process is guided by QuaTEx expressions, which dictate the execution of new steps as indicated by their # operators until a final value is computed. Upon completing a batch of simulations, the tool assesses the current confidence level using the Student's t-distribution to determine if further simulations are warranted.

Consider the QuaTEx expression below. The rval operation is used to access the state of the system under analysis. It takes as argument a string specifying the expression to be evaluated. The "steps" expression is a predefined one that returns the number of rewriting steps. If in the expression there is a variable of the module it is instantiated by its current value along the simulation. Thus,

the expression "State |= machine Client at NoMoneyToWithDraw" makes use of the satisfaction predicate and the atomic proposition machine_at_ introduced in Section 4.3. Specifically, the expression calculates the number of steps until one of the clients of the system is in state NoMoneyToWithDraw.

```
StepsOne() =
        if s.rval("steps") > 200
        then 200
        else if (s.rval("State |= machine Client at NoMoneyToWithDraw"))
             then s.rval("steps")
             else # StepsOne()
             fi
        fi;
eval E[StepsOne()];
```

The statistical model checker can be invoked with the following command, where client-server-mc is the name of the file in which the above module CLIENT-SERVER-CHECK is. The file formula.quatex contains the previous QuaTEx expression.

```
% umaudemc scheck –j 0 client–server–mc 'execute(tcTwoClients, init)' formula.quatex –a 0.1
Number of simulations = 44220
  μ = 106.49823609226594 σ = 63.90298547450405 r = 0.4998597119551526
```

The command was executed specifying the confidence interval. More precisely, the -a or –alpha flag allows the specification of its complement: $1 - \alpha$ is known as the confidence level of the interval. The -j or –jobs flag was used to specify the number of parallel simulation processes to use — the value 0 indicates to start as many jobs as CPU units in the machine. The result shows that the expected number of steps is 106.50, with a radius for the confidence interval σ of 63.90. The execution took place in around 300 seconds.

4.5. Other case studies

In addition to the Client-Server example, two additional case studies are presented in this section. As already pointed out, other case studies may be found in the companion web site. Section 4.5.1 uses another example from P's tutorial, a model of a simple reactive systems. Section 4.5.2 uses an example from [18].

4.5.1. An espresso machine

In this example, an espresso machine is modeled using two machines, namely the coffee machine itself and its control panel. Users are then modeled as test cases that interact with the coffee machine through its control panel. The control panel provides an interface for the user to perform actions such as resetting the machine, turning the steamer on or off, requesting an espresso, and emptying the grounds by opening the container. It receives these user inputs and sends corresponding commands to the coffee maker.

Fig. 9 shows the EspressoCoffeMaker, CoffeMakerControlPanel, and User state machines. The control panel begins in a startup state, where it initiates the coffee maker's warm-up process. Once warming is complete, it transitions to the ready state, allowing operations such as brewing coffee or starting the steamer. When a coffee request is received, the machine first grinds the beans before proceeding to brew. At any stage, if an error occurs –such as a lack of water or beans– the control panel alerts the user and transitions to an error state, where it awaits a reset from the user. The User state machine simply operates the machine to prepare two cups of coffee, responding to the eventual occurrences of error messages.

As for the Client-Server example, the first step is to use the compiler to automatically generate a Maude system module from the P source files of the example. This module can be used to execute the system, although the output is not very interesting, the user gets its two cups and the program terminates. But it is not necessarily always the case. The system may keep producing errors forever and never terminate. Or it may be wrongly implemented. As in previous sections, some properties on the resulting specification are analyzed.

For example, Maude's search command may be used, bounded by a maximum depth of 100 rewrites, to check that there are no deadlocks other than when the user gets its two cups.

```
Maude> search [,100]
          execute(tcSaneUserUsingCoffeeMachine, init)
          =>! SES:SystemExecState .
```

Several solutions are provided, since the cups may be produced after a number of errors, which would produce error messages. An abstraction of the system making the state space reachable from the initial state finite may consist of removing the error messages in the log of the system (i.e., a projection of the state).

```
mod ESPRESSO–MACHINE–ABSTRACTION is
  inc ESPRESSO–MACHINE–RUN .

  var ES : ExecState .
  vars S I : Nat .
  var Sys : System .
  var Log : String .
```

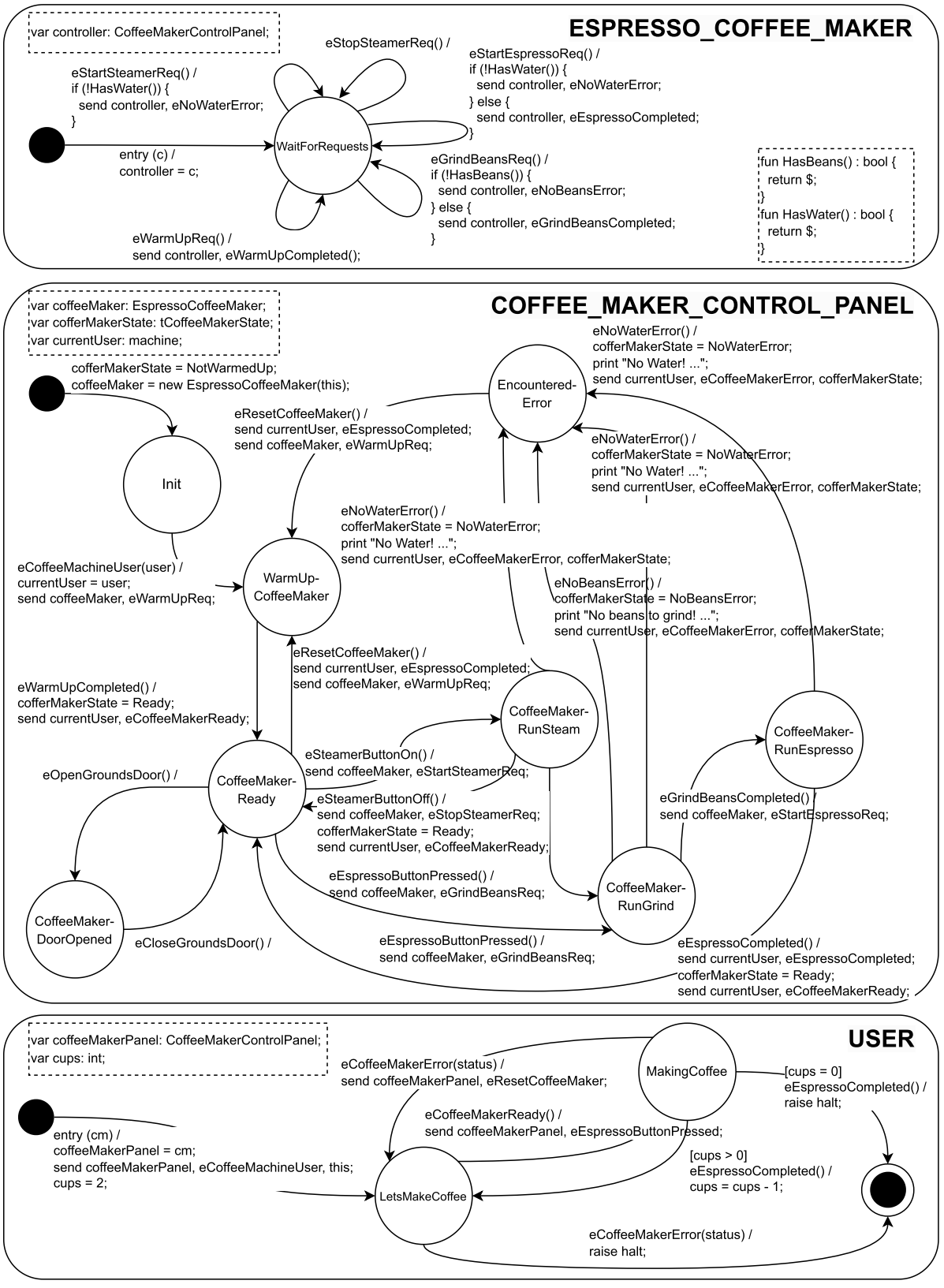


Fig. 9. The EspressoCoffeMaker, CoffeMakerControlPanel, and User state machines.

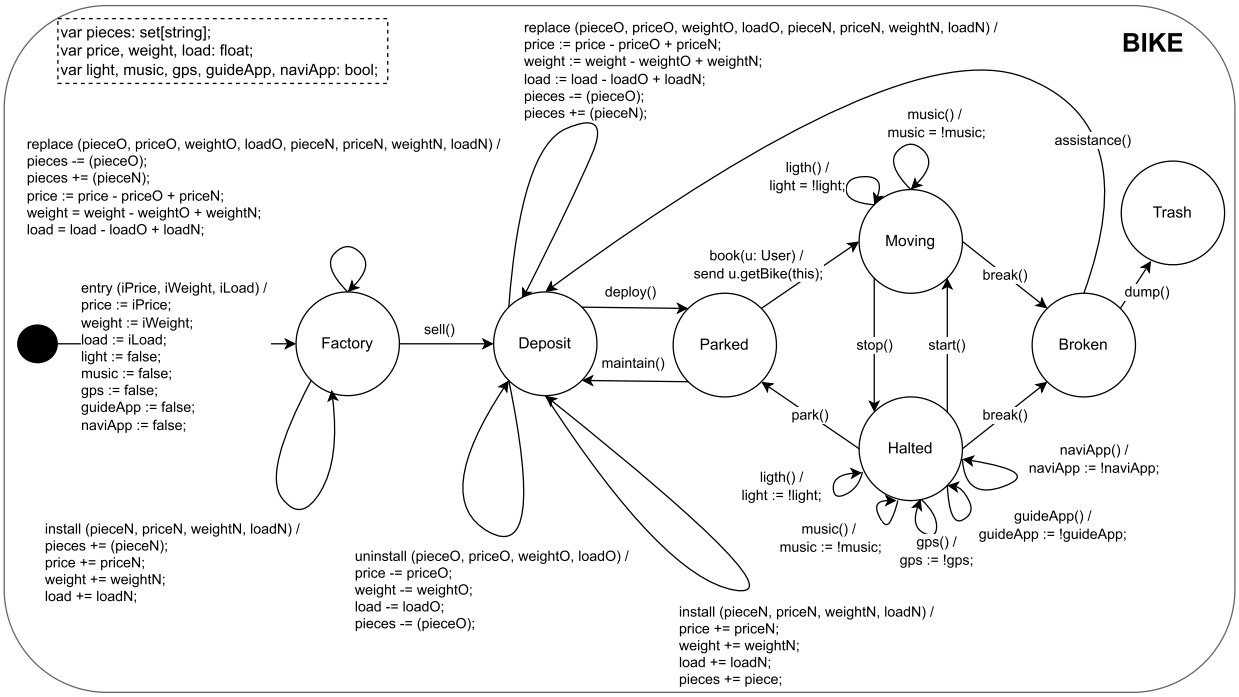


Fig. 10. The Bike state machine.

```
ceq [abstraction–log] : [ES, S, I, Sys, Log]
   = [ES, S, I, Sys, ""]
   if Log =/= "" .
endm
```

With this abstraction, a search task can be executed without a bound, producing one single solution.

```
Maude> search
          execute(tcSaneUserUsingCoffeeMachine, init)
          =>! SES:SystemExecState .
```

The command visits 136 states, which makes the abstracted version of the system quite small. Once all the requirements for model checking are met, other interesting properties can be checked. For example, consider the following proposition:

```
op machine_gets_at_ : MachId EventName Name –> Prop .
```

The term machine MId gets EN at St represents a machine with identifier MId, with incoming event EN in the head of its event queue at state St. Then, it can be checked that it is always the case that when a coffee cup is requested, it is eventually provided.

```
red modelCheck(execute(tcSaneUserUsingCoffeeMachine, init),
      [] (machine id(1) gets eEspressoButtonPressed at CoffeeMakerReady
          –> <> machine id(1) gets eEspressoCompleted at CoffeeMakerRunEspresso)) .
rewrites: 2541 in 25ms cpu (26ms real) (97866 rewrites/second)
result Bool: true
```

4.5.2. A bike sharing system

This case study was originally presented in [18]. It is based on a bike sharing system inspired in a case study originally presented in [54]. The entire code of the case study and the properties to be checked are available at https://github.com/PST-P/p-pst.

The model consists of a state machine for a bike and four ghost machines representing its environment. Fig. 10 shows the Bike machine and Fig. 11 the environment machines. The state machines cover the lifecycle bikes, including their manufacturing, maintenance, and use.

Starting from a base bike, a number of optional devices can be installed during its manufacturing, and later be replaced, removed, or added during maintenance at the deposit. Each one of these components has a price and a weight, and consumes certain amount of energy. The program makes use of local functions Handle to handle the installation, replacement, and removal of accessories on the bikes.

The example uses several data structures and illustrates many of the different features of P. For example, a bike keeps its components in the variable pieces of type set[string], and a factory manager keeps information on bike components in the variables price, weight, and load, all of type map[string,float].

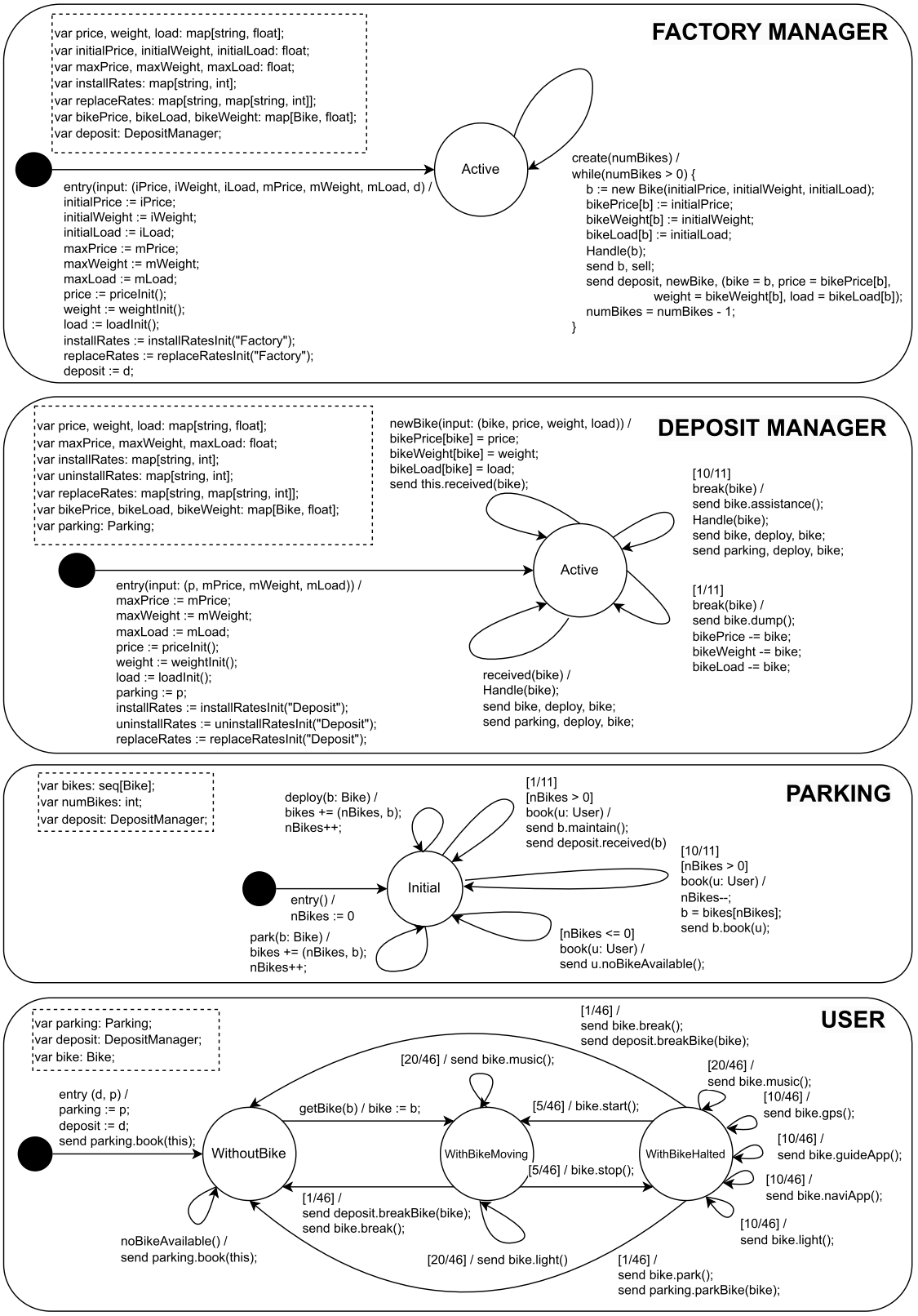


Fig. 11. Ghost machines.

Listing 1: Ratio between Moving states and all states.

```
PercBikeMoving() = PercBikeMovingAux(0, 0);
PercBikeMovingAux(all, moving)
  = if (s.rval("alive(id(9), State)"))
    then if (s.rval("state(id(9), Broken, State)"))
         then moving / all
         else if (s.rval("state(id(9), Moving, State)"))
              then # PercBikeMovingAux(all + 1, moving + 1)
              else # PercBikeMovingAux(all + 1, moving)
              fi
         fi
    else # PercBikeMovingAux(all, moving)
    fi;

eval E[PercBikeMoving()];
```

Listing 2: Number of times a bicycle is used before going to trash.

```
BikeLive() = BikeLiveAux(0);
BikeLiveAux(count)
  = if (s.rval("alive(id(9), State)"))
    then if (s.rval("state(id(9), Trash, State)"))
         then count
         else # BikeLiveAux(count + 1)
         fi
    else # BikeLiveAux(count)
    fi;

eval E[BikeLive()];
```

The different behaviors of ghost machines may be described as the specification of any possible behavior. In this case study, they have been modeled as a probabilistic system, so that all decisions are quantified. The example illustrates how, while some actions depend on the reception of given events, other are probabilistic, with fixed probabilities or with probabilities that depend on the state of the machines with respect to the given constraints.

- For example, when a user is in the WithoutBike state, she/he can receive a noBikeAvailable() or a bike(b) event, with a bike b, firing one transition or the other: the decision only depends on the event that is received.
- Upon the reception of an event, or even spontaneously without the need for an event, multiple transitions may be fired depending on a probability value. For instance, a user can do several things at the WithBikeMoving state: with probability 20/46, it can send a music() event to the bike (to turn the music on or off); with probability 5/46, it can send a stop() event to the bike; etc.
- There can also be conditions associated with transitions, so that an event may fire one or the other depending on the evaluation of the condition. For example, in the Parking state machine, the reception of a book() event in the Initial state will result in one transition or another, executing the corresponding actions, depending on whether there are bikes available or not ([nBikes > 0] or [nBikes <= 0]).
- In the case of the FactoryManager and DepositManager state machines, the probabilities depend on the state of the bikes under operation and the given constrains. It is all handled in the Handle functions, which are not useful for the current goals, but all decisions are taken following some probabilities and given some constraints (see [18] for details).

Consider the two QuaTEx expressions shown in Listings 1 and 2. Listing 1 estimates the ratio between the number of times the machine is in the Moving state and the total number of states visited. The PercBikeMovingAux function recursively calculates the total number of visited states and the number of times in the Moving state. When the state Broken is visited, it returns the value of the quotient. The result shows that the bike is in the Moving state 34.48% of the time, with a confidence interval (CI), computed using the Student's t-test, of 0.019. Listing 2 estimates the useful life of a bike. Specifically, it counts the number of times it is in states other than the Trash state. In fact, it counts the number of steps. The result shows that the bike takes an average of 9496.184 steps, with CI of 817.17, computed also using the Student's t-test. This is a property with a high variability due to the small probability of occurrence of the dump event: the calculated variance is 7066.65.

This section ends by comparing the results provided by the proposed approach (i.e., using $\mathcal{R}_{\mathsf{P}}$ with umaudemc), and the approach proposed in [18] with P's official interpreter plus MultiVeStA. Before the comparison, two things must be said on the differences between these two tools. The first is that, even though both use QuaTEx to formulate the expressions to evaluate, the queries are carried out quite differently. Whilst in P + MultiVeStA the queries are fixed, as provided by the tool, umaudemc allows the use of any user-defined Boolean or numerical expression. In the former, queries are limited to checking the number of steps executed, if the execution has terminated, if a specific machine is alive, if a machine is in a given state, or if a machine has received a given event. The second difference is in the steps count. While P + MultiVeStA counts as a step each instruction in the code, $\mathcal{R}_{\mathsf{P}}$ + umaudemc

Listing 3: Times a bicycle gets to the Moving state before getting into the Broken state.

```
Moving() = MovingAux(0.0, 0.0);

MovingAux(current, moving)
 = if (s.rval("alive(id(9), State)"))
    then if (s.rval("state(id(9), Factory, State)"))
            then MovingAuxAux(current, 1.0, moving)
            else if (s.rval("state(id(9), Deposit, State)"))
                    then MovingAuxAux(current, 2.0, moving)
                    else if (s.rval("state(id(9), Parked, State)"))
                            then MovingAuxAux(current, 3.0, moving)
                            else if (s.rval("state(id(9), Moving, State)"))
                                    then MovingAuxAux(current, 4.0, moving)
                                    else if (s.rval("state(id(9), Halted, State)"))
                                            then MovingAuxAux(current, 5.0, moving)
                                            else if (s.rval("state(id(9), Broken, State)"))
                                                    then moving // exit point
                                                    else if (s.rval("state(id(9), Trash, State)"))
                                                            then MovingAuxAux(current, 7.0, moving)
                                                            else # MovingAux(current, moving)
                                                            fi
                                                    fi
                                            fi
                                    fi
                            fi
                    fi
            fi
    else # MovingAux(current, moving)
    fi;

MovingAuxAux(previous, current, moving)
  = if (previous == current)
    then # MovingAux(current, moving)
    else if (s.rval("state(id(9), Moving, State)"))
            then # MovingAux(4.0, moving + 1)
            else # MovingAux(current, moving)
            fi
    fi;

eval E[Moving()];
```

counts nondeterministic events. For these reasons, a simple expression that only uses the type of queries in P + MultiVeStA, and that does not depend on the way steps are counted, is used to compare the results provided by these two approaches.

The expression in Listing 3 estimates the number of times one of the bikes gets into the Moving state before getting to the Broken state for the first time. Note that the previous expressions were calculating, respectively, the number of steps in the `Moving` state and the ratio between the number of steps in the `Moving` state and the number of steps in any other state. In this expression, the previous state is passed as an argument, and the increment only takes place if the states are different. In this way, it can be identified how many times a bike arrives to Moving coming from a different state. To do this, states are numbered (Init is 0.0, Factory is 1.0, and so on). The MovingAux function takes as argument the current state number and the number of times the Moving state has been visited. It either returns the counter if the Broken state is visited or invokes another auxiliary function MovingAuxAux. This second auxiliary function gets the previous and current state numbers, and depending on whether they are the same or different, and whether the current state is the Moving one, it returns a different value.

After 300 simulations, the results are quite similar. The P + MultiVeStA approach gives an expected mean of 3.467, with variance 9.307 and CI/2 0.454. The $\mathcal{R}_{\mathsf{P}}$ + umaudemc approach computes a mean of 3.587, with variance 3.203 and CI/2 0.479. Even though the variances are different, possibly due to the limitation in the number of simulations to 300, the experiments show very similar means, with similar confidence intervals. This is also backed with more experiments that are not presented in this section.

5. Correspondence between $\mathcal{R}_{\mathsf{P}}$ + umaudemc and P + MultiVeStA

This section discusses the question of correspondence between P's official and $\mathcal{R}_{\mathsf{P}}$ rewriting logic semantics from the viewpoint of simulation and analysis, including how statistical model checking (SMC) is employed for quantitative analysis of P programs.

The P programming language provides command-line tools for simulating and verifying programs. After compilation, the '`p check`' command systematically explores program behaviors based on defined test cases. This command takes as input the name of a test case and the number of *schedules* (simulations) to explore during execution. Test cases are defined within the P program itself. Additionally, '`p check`' can accept a *seed* for pseudo-random number generation; if this parameter is not provided, a random seed is generated internally by the interpreter.

A schedule defines a specific sequence of events and state transitions that the *scheduler* executes to control program behavior. It determines the precise order in which machines (or agents) process events, influencing the system's overall behavior. The schedule specifies which machine is active at each step, managing the interleaving of actions between machines, which may have pending events that could be processed in various orders. The scheduler in P incorporates controlled randomness to simulate nondeterministic choices using a random number generator with the specified or internally generated seed. Various schedules can be explored to simulate different execution paths without requiring multiple executions in uncontrolled environments. The number of schedules explored corresponds to the value specified in the '`p check`' command.

The scheduler constructs a schedule as follows. In a given state s, it identifies the number n of possible one-step transitions; it then selects one of these n transitions uniformly at random to transition to the next state s'. This dynamic approach, known as Monte Carlo simulation with uniform sampling, continues until no transitions are available or the desired depth limit is reached. During testing or verification, the interpreter can execute a program under multiple schedules to ensure comprehensive coverage of possible execution paths, helping identify behaviors that emerge under specific event sequences.

Statistical model checking (SMC) was introduced to P programs in [18]. This approach integrates MultiVeStA [53] with the P interpreter, enabling MultiVeStA to monitor the advancement of schedules created by the interpreter. MultiVeStA performs incremental analysis of QuaTEx formulas based on the execution progress of P programs. The completeness of this approach relies on the fairness of P's scheduler in constructing schedules and the number of runs required by MultiVeStA to achieve statistical significance.

This paper presents an alternative approach, extending the work in [19], with notable differences. The rewriting logic semantics $\mathcal{R}_{\mathsf{P}}$ serves as a formal interpreter for the P programming language, eliminating reliance on P's infrastructure for program execution. Instead, execution occurs within the Maude system, where the target P program is loaded and executed using $\mathcal{R}_{\mathsf{P}}$. In P's terminology, Maude also utilizes schedules to simulate program execution and perform state-space exploration. The primary distinction lies in the method of constructing these schedules: Maude's schedules are not generated using controlled randomness, but rather through arbitrary choice within the Maude system.

The lack of controlled randomness in Maude poses a challenge for SMC due to unquantified nondeterminism in selecting among multiple transition choices from a given state. To address this, umaudemc is integrated with Maude in a manner similar to the integration of MultiVeStA with the P interpreter in [18]. At each execution state s, umaudemc determines the number n of possible one-step transitions and uniformly selects one to guide Maude's execution, ultimately constructing a "fair" schedule. This approach aligns with the P interpreter's method for simulating concurrent behavior with Monte Carlo simulations. It is noteworthy that umaudemc can employ different probability distributions for transition sampling; however, in this work, only the uniform distribution is used.

Proving that two language semantics are equivalent is a significant challenge, especially for a concurrent language like P. Beyond the similarities in their concurrency models discussed above, it is crucial to consider how transitions are "axiomatized" in each semantic framework. Recall that $\mathcal{R}_{\mathsf{P}}$ was developed primarily using P's user documentation, and the source code of its compiler and interpreter. Presenting a formal proof of equivalence is beyond the scope of this work. Instead, the new verification and validation capabilities afforded to P by the Maude ecosystem are leveraged here to extensively test both the proposed semantics and the specifications of interest. This approach, while not conclusive or fullproof, does provide empirical support for the claim that the semantics are likely equivalent from an exhaustive and practical standpoint rather than through formal proof. Further research in this area is essential (see Section 6) and complementary methods, such as differential software testing [35], could be valuable.

6. Related work and concluding remarks

To the best of the authors' knowledge, no tooling is available for P other than what is reported in the official release of P, for execution and bounded model checking (or systematic testing), for statistical model checking [18], and in the previous sections. However, there is substantial related work on using rewriting logic as a semantic framework. In this context, executable specifications in Maude are employed to provide semantics to different languages, which is the approach taken in this work. The following paragraphs provide a glimpse on using rewriting logic (and Maude) as a semantic framework.

An operational semantics of P is provided in [13]. There, 22 transition rules are given: 10 for statement execution, 8 for event handling, and 4 for error transitions. The rewriting semantics developed and introduced in the present work is over 2000 lines of Maude code, providing not only a semantics for most of the features of the language, but also an interpreter for it. Most of the language constructions are covered, including its key elements for concurrency and nondeterminism; however, other elements, such as monitors and external types and functions are not part of $\mathcal{R}_{\mathsf{P}}$. As above explained, this proposed semantics is also useful in practice for the discussed purposes.

Rewriting logic has been demonstrated to be a highly effective logical and semantic framework [34] for expressing and giving semantics to a wide range of systems, including models of concurrency, distributed algorithms, network protocols, and programming language semantics (see [37,39,41,9,43,42]). For a comparison of rewriting logic and other frameworks for defining programming languages, specifically with Structural Operational Semantics, see [42]. The rewriting logic semantics approach has been used to define several programming languages or large fragments of them, e.g., C [24], Scheme [36], Java 1.4 [25,27], and NASA's Plan Execution Language (PLEXIL) [14]. C's rewriting logic semantics was latter used to add model checking support for C programs [24].

The rewriting logic semantics of AADL was used for model checking, as reported in [6,44]. A wide range of real-time Domain-Specific Visual Languages (DSVLs), as well as their dynamic real-time behavior, can be specified with a rigorous semantics using the e-Motions framework [47] and MOMENT2's support for real-time DSVLs [7]. The development of a tool for statistical model checking for e-Motions DSVLs was reported in [17].

This paper introduced the rewriting logic semantics $\mathcal{R}_{\mathsf{P}}$ for the P programming language. Besides being a formal interpreter of the language, $\mathcal{R}_{\mathsf{P}}$ can be used for automated analyses beyond the reach of current techniques and tools available to P programs. These analyses encompass state-space exploration, LTL model checking, and statistical model checking, the latter being achievable by augmenting $\mathcal{R}_{\mathsf{P}}$ with probabilities for the concurrent transitions. Additionally, formal verification techniques, such as equational abstractions of P programs, were presented in conjunction with $\mathcal{R}_{\mathsf{P}}$. The effectiveness of these capabilities was demonstrated with the specification and analysis of several case studies, including a running example borrowed from P's official tutorial. Several of these techniques, notably LTL model checking, represent novel contributions to the analysis of P programs. The overarching aim is to provide the rewriting logic semantics $\mathcal{R}_{\mathsf{P}}$ as a valuable resource for the P community. It can serve as both a platform for formal analysis, and an experimental playground for exploring new language features and enhancements.

There is still significant work to be done regarding more comprehensive analyses of case studies, including other challenging ones. In particular, the authors plan to support foreign types and functions through Maude interfaces [49], and the various mechanisms available for the specification of open systems in Maude [22]. Extending the semantics to these and other currently unsupported features is essential for enhancing the practical impact of the contributions outlined in this paper. Furthermore, this extension has the potential to pave the way for identifying promising new methodologies for formally verifying P programs. As is customary when establishing a rewriting logic semantics for a language like P, a variety of techniques and tools from the Maude Formal Environment become accessible for program verification across various forms. It would be beneficial to delve deeper into exploring advanced abstraction techniques [5], symbolic methods such as symbolic reachability analysis with rewriting modulo SMT [48,15,16] and narrowing [15,30], and their combination [33]. Additionally, there is potential for extending these efforts to encompass deductive and inductive theorem proving approaches. Finally, further work is needed to validate the correspondence between P's official semantics and $\mathcal{R}_{\mathsf{P}}$, maybe, with the help of techniques such as differential software testing [35].

CRediT authorship contribution statement

Francisco Durán: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Carlos Ramírez:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Investigation, Formal analysis, Data curation. **Camilo Rocha:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Nicolás Pozas:** Writing – review & editing, Writing – original draft, Validation, Software, Resources, Investigation, Formal analysis, Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to thank the reviewers for carefully reading the manuscript; their comments have been of great help in improving its quality and clarity. This work was partially supported by Amazon Research Awards (Fall 2021) Project "Probabilistic and Symbolic Tools for P Program Verification". F. Durán and N. Pozas have been partially supported by projects TED2021-130666B-I00 and PID2021-125527NB-I00 funded by the Spanish government. C. Ramírez and C. Rocha have been partially supported by the SGR project PROMUEVA (BPIN 2021000100160) supervised by Minciencias.

References

[1] G. Agha, Actors: a Model of Concurrent Computation in Distributed Systems (Parallel Processing, Semantics, Open, Programming Languages, Artificial Intelligence), PhD thesis, University of Michigan, USA, 1985.

[2] G. Agha, J. Meseguer, K. Sen, PMaude: rewrite-based specification language for probabilistic object systems, in: Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005), Electron. Notes Theor. Comput. Sci. 153 (2) (2006) 213–239.

[3] M. AlTurki, J. Meseguer, PVeStA: a parallel statistical model checking and quantitative analysis tool, in: A. Corradini, B. Klin, C. Cîrstea (Eds.), Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings, in: Lecture Notes in Computer Science, vol. 6859, Springer, 2011, pp. 386–392.

[4] T. Andrews, S. Qadeer, S.K. Rajamani, J. Rehof, Y. Xie, Zing: a model checker for concurrent software, in: Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings, in: Lecture Notes in Computer Science, vol. 3114, Springer, 2004, pp. 484–487.

[5] K. Bae, J. Meseguer, Predicate abstraction of rewrite theories, in: G. Dowek (Ed.), Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings, in: Lecture Notes in Computer Science, vol. 8560, Springer, 2014, pp. 61–76.

[6] K. Bae, P.C. Ölveczky, A. Al-Nayeem, J. Meseguer, Synchronous AADL and its formal analysis in real-time maude, in: S. Qin, Z. Qiu (Eds.), Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings, in: Lecture Notes in Computer Science, vol. 6991, Springer, 2011, pp. 651–667.

[7] A. Boronat, P.C. Ölveczky, Formal real-time model transformations in MOMENT2, in: D.S. Rosenblum, G. Taentzer (Eds.), Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, in: Lecture Notes in Computer Science, vol. 6013, Springer, 2010, pp. 29–43.

[8] A. Bouhoula, J.-P. Jouannaud, J. Meseguer, Specification and proof in membership equational logic, Theor. Comput. Sci. 236 (1) (2000) 35–132.
[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350, Springer, 2007.
[10] M. Clavel, M. Palomino, A. Riesco, Introducing the ITP tool: a tutorial, J. Univers. Comput. Sci. 12 (11) (2006) 1618–1650.
[11] P. Deligiannis, A. Senthilnathan, F. Nayyar, C. Lovett, A. Lal, Industrial-strength controlled concurrency testing for C# programs with coyote, in: Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, in: Lecture Notes in Computer Science, vol. 13994, Springer, 2023, pp. 433–452.
[12] A. Desai, V. Gupta, E.K. Jackson, S. Qadeer, S.K. Rajamani, D. Zufferey, P: safe asynchronous event-driven programming, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16–19, 2013, ACM, 2013, pp. 321–332.
[13] A. Desai, S. Qadeer, P: modular and safe asynchronous programming, in: S.K. Lahiri, G. Reger (Eds.), Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings, in: Lecture Notes in Computer Science, vol. 10548, Springer, 2017, pp. 3–7.
[14] G. Dowek, C.A. Muñoz, C. Rocha, Rewriting logic semantics of a plan execution language, in: B. Klin, P. Sobocinski (Eds.), Proceedings Sixth Workshop on Structural Operational Semantics, SOS 2009, Bologna, Italy, August 31, 2009, in: EPTCS, vol. 18, 2009, pp. 77–91.
[15] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, C.L. Talcott, Programming and symbolic computation in Maude, J. Log. Algebraic Methods Program. 110 (2020).
[16] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, C.L. Talcott, Equational unification and matching, and symbolic reachability analysis in Maude 3.2 (system description), in: Automated Reasoning - 11th International Joint Conference, IJCAR, Proceedings, in: Lecture Notes in Computer Science, vol. 13385, Springer, 2022, pp. 529–540.
[17] F. Durán, A. Moreno-Delgado, J.M. Álvarez-Palomo, Statistical model checking of e-motions domain-specific modeling languages, in: P. Stevens, A. Wasowski (Eds.), Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, the Netherlands, April 2-8, 2016, Proceedings, in: Lecture Notes in Computer Science, vol. 9633, Springer, 2016, pp. 305–322.
[18] F. Durán, N. Pozas, C. Ramírez, C. Rocha, Statistical model checking for P, in: Formal Methods for Industrial Critical Systems - 28th International Conference, FMICS, Proceedings, in: Lecture Notes in Computer Science, vol. 14290, Springer, 2023, pp. 40–56.
[19] F. Durán, N. Pozas, C. Ramírez, C. Rocha, Towards a rewriting-logic semantics of P, in: L. Panizo (Ed.), Actas de las XXII Jornadas sobre Programación y Lenguajes (PROLE 2023), Sistedes, 2023.
[20] F. Durán, C. Rocha, J.M. Álvarez, Tool interoperability in the Maude formal environment, in: A. Corradini, B. Klin, C. Cîrstea (Eds.), Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings, in: Lecture Notes in Computer Science, vol. 6859, Springer, 2011, pp. 400–406.
[21] F. Durán, C. Rocha, J.M. Álvarez, Towards a Maude formal environment, in: G. Agha, O. Danvy, J. Meseguer (Eds.), Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday, in: Lecture Notes in Computer Science, vol. 7000, Springer, 2011, pp. 329–351.
[22] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, C. Talcott, Programming open distributed systems in Maude, in: 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024, September 09–11, 2024, Milano, Italy, ACM, 2024.
[23] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, in: WRLA2002, in: Electronic Notes in Theoretical Computer Science, vol. 71, Elsevier, 2002, pp. 162–187.
[24] C. Ellison, G. Rosu, An executable formal semantics of C with applications, in: J. Field, M. Hicks (Eds.), Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012, ACM, 2012, pp. 533–544.
[25] A. Farzan, F. Chen, J. Meseguer, G. Rosu, Formal analysis of Java programs in JavaFAN, in: R. Alur, D.A. Peled (Eds.), CComputer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings, in: Lecture Notes in Computer Science, vol. 3114, Springer, 2004, pp. 501–505.
[26] A. Farzan, J. Meseguer, State space reduction of rewrite theories using invisible transitions, in: Algebraic Methodology and Software Technology, 11th International Conference, AMAST, Proceedings, in: Lecture Notes in Computer Science, vol. 4019, Springer, 2006, pp. 142–157.
[27] A. Farzan, J. Meseguer, G. Rosu, Formal JVM code analysis in JavaFAN, in: C. Rattray, S. Maharaj, C. Shankland (Eds.), Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings, in: Lecture Notes in Computer Science, vol. 3116, Springer, 2004, pp. 132–147.
[28] C. Hensel, S. Junges, J. Katoen, T. Quatmann, M. Volk, The probabilistic model checker Storm, Int. J. Softw. Tools Technol. Transf. 24 (4) (2022) 589–610.
[29] C. Hewitt, P.B. Bishop, R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Standford, CA, USA, August 20-23, 1973, William Kaufmann, 1973, pp. 235–245.
[30] B. Kang, K. Bae, Symbolic reachability analysis of distributed systems using narrowing and heuristic search, in: C. Artho, P.C. Ölveczky (Eds.), Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS 2022, Auckland, New Zealand, 7 December 2022, ACM, 2022, pp. 34–44.
[31] M.Z. Kwiatkowska, G. Norman, D. Parker, The PRISM benchmark suite, in: Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012, IEEE Computer Society, 2012, pp. 203–204.
[32] A. Legay, B. Delahaye, S. Bensalem, Statistical model checking: an overview, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G.J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (Eds.), Runtime Verification - First International Conference, RV. Proceedings, in: Lecture Notes in Computer Science, vol. 6418, Springer, 2010, pp. 122–135.
[33] R. López-Rueda, S. Escobar, J. Sapiña, An efficient canonical narrowing implementation with irreducibility and SMT constraints for generic symbolic protocol analysis, J. Log. Algebraic Methods Program. 135 (2023) 100895.
[34] N. Martí-Oliet, J. Meseguer, Rewriting logic as a logical and semantic framework, in: D. Gabbay, F. Guenthner (Eds.), Handbook of Philosophical Logic, vol. 9, second edition, Kluwer Academic Publishers, 2002, pp. 1–87.
[35] W.M. McKeeman, Differential testing for software, Digit. Tech. J. 10 (1) (1998) 100–107.
[36] P.O. Meredith, M. Hills, G. Rosu, A K definition of Scheme, Technical report, 2007.
[37] J. Meseguer, Rewriting as a unified model of concurrency, in: CONCUR'90, Theories of Concurrency: Unification and Extension, Proceedings, in: Lecture Notes in Computer Science, vol. 458, Springer, 1990, pp. 384–400.
[38] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1) (1992) 73–155.
[39] J. Meseguer, N. Martí-Oliet, From abstract data types to logical frameworks, in: Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, Selected Papers, in: Lecture Notes in Computer Science, vol. 906, Springer, 1994, pp. 48–80.
[40] J. Meseguer, M. Palomino, N. Martí-Oliet, Equational abstractions, Theor. Comput. Sci. 403 (2–3) (2008) 239–264.
[41] J. Meseguer, G. Roşu, Rewriting logic semantics: from language specifications to formal analysis tools, in: Automated Reasoning - Second International Joint Conference, IJCAR, Proceedings, in: Lecture Notes in Computer Science, vol. 3097, Springer, 2004, pp. 1–44.
[42] J. Meseguer, G. Roşu, The rewriting logic semantics project: a progress report, Inf. Comput. 231 (2013) 38–69.
[43] J. Meseguer, G. Roşu, The rewriting logic semantics project, Theor. Comput. Sci. 373 (3) (2007) 213–237.
[44] S.P. Miller, D.D. Cofer, L. Sha, J. Meseguer, A. Al-Nayeem, Implementing logical synchrony in integrated modular avionics, in: 2009 IEEE/AIAA 28th Digital Avionics Systems Conference, 2009, 1.A.3.

[45] NuITP Developers, NuITP's web site, https://nuitp.webs.upv.es/, 2024. (Accessed 7 March 2024).
[46] P. Developers, P's web site, https://p-org.github.io/P/, 2024. (Accessed 7 March 2024).
[47] J.E. Rivera, F. Durán, A. Vallecillo, Formal specification and analysis of domain specific models using maude, Simulation 85 (11–12) (2009) 778–792.
[48] C. Rocha, J. Meseguer, C.A. Muñoz, Rewriting modulo SMT and open system analysis, J. Log. Algebraic Methods Program. 86 (1) (2017) 269–297.
[49] R. Rubio, Maude as a library: an efficient all-purpose programming interface, in: K. Bae (Ed.), Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 13252, Springer, 2022, pp. 274–294.
[50] R. Rubio, N. Martí-Oliet, I. Pita, A. Verdejo, Strategies, model checking and branching-time properties in maude, J. Log. Algebraic Methods Program. 123 (2021) 100700.
[51] R. Rubio, N. Martí-Oliet, I. Pita, A. Verdejo, Model checking strategy-controlled systems in rewriting logic, Autom. Softw. Eng. 29 (1) (2022) 7.
[52] R. Rubio, N. Martí-Oliet, I. Pita, A. Verdejo, QMaude: quantitative specification and verification in rewriting logic, in: M. Chechik, J. Katoen, M. Leucker (Eds.), Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings, in: Lecture Notes in Computer Science, vol. 1400, Springer, 2023, pp. 240–259.
[53] S. Sebastio, A. Vandin, Multivesta: statistical model checking for discrete event simulators, in: ValueTools '13, Brussels, BEL, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.
[54] M.H. ter Beek, A. Legay, A. Lluch-Lafuente, A. Vandin, Statistical analysis of probabilistic models of software product lines with quantitative constraints, in: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, ACM, 2015, pp. 11–15.
[55] A. Vandin, D. Giachini, F. Lamperti, F. Chiaromonte, MultiVeStA: statistical analysis of economic agent-based models by statistical model checking, in: J. Bowles, G. Broccia, R. Pellungrini (Eds.), From Data to Models and Back - 10th International Symposium, DataMod 2021, Virtual Event, December 6-7, 2021, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 13268, Springer, 2021, pp. 3–6.