DEVS Modeling and Simulation Using BDI Agents: Preliminary Considerations

Amelia Bădică
University of Craiova
Craiova, Romania
ameliabd@yahoo.com

Costin Bădică
University of Craiova
Craiova, Romania
cbadica@software.ucv.ro

Ion Buligiu
University of Craiova
Craiova, Romania
buligiu.ion@gmail.com

Liviu Ciora
University of Craiova
Craiova, Romania
liviuciora2004@yahoo.com

ABSTRACT

In this paper we propose a method for mapping DEVS simulation models to BDI multi-agent systems. Our work opens the possibility of reusing BDI multi-agent frameworks for the modeling and simulation of discrete event dynamic systems following the DEVS formalized approach. Moreover, thanks to the key features of the BDI architecture, we are are now able to systematically enhance simulation models with cognitive and intelligence aspects specific to BDI agents. Our method is demonstrated with the help of a simple, yet illustrative example that we developed using the Jason multi-agent systems development platform.

CCS CONCEPTS

• **Computing methodologies → Multi-agent systems**; **Modeling and simulation**; **Systems theory**; • **Software and its engineering → Simulator / interpreter**;

KEYWORDS

multi-agent system, belief-desire-intention, modeling and simulation, discrete event dynamic system

ACM Reference Format:
Amelia Bădică, Costin Bădică, Ion Buligiu, and Liviu Ciora. 2018. DEVS Modeling and Simulation Using BDI Agents: Preliminary Considerations. In *WIMS '18: 8th International Conference on Web Intelligence, Mining and Semantics, June 25–27, 2018, Novi Sad, Serbia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3227609.3227681

1 INTRODUCTION

The aim of this paper is to introduce our proposed method for mapping DEVS simulation models to BDI multi-agent systems. Our results pave the way for reusing BDI-based multi-agent frameworks for the modeling and simulation of discrete event dynamic systems following the DEVS formalized approach [21]. In particular, thanks to the specific features of the BDI multi-agent architecture, we are are now able to systematically enhance simulation models with cognitive, intelligence and social aspects of BDI agents [10]. Our proposal is demonstrated by considering a simple, yet illustrative example that we developed with the help of the Jason multi-agent systems development platform [4].

The basic paths followed for modeling and simulating of complex dynamic systems take into account the nature of time, as well as the nature of system variables, i.e. both can be either continuous or discrete [19]. In this paper we are interested in the *discrete event systems specification*, also known as DEVS, that has a long tradition in the field of modeling and simulation of discrete event dynamic systems [21].

According to DEVS, the system dynamics is defined by a set of timed discrete internal and external events that determine the instantaneous update of the system state. The DEVS formalism is quite general and was applied to many problems and systems in various domains. Moreover, this formalism has a quite strong and rigorous background that enabled the development of provable correct simulation algorithms, currently supported by many practical DEVS-based modeling and simulation tools [18].

A special feature of the DEVS modeling framework is that it supports a hierarchically modular decomposition of the system model with a clean separation of the system modules from the simulation engine. DEVS modules (also sometimes called DEVS models) are independent, possibly autonomous entities, that communicate by exchanging messages, similarly to autonomous agents.

This analogy between DEVS modules and autonomous agents immediately raises the question of investigating the existing relationships between DEVS and multi-agent frameworks. In particular, *belief-desire-intention*, also known as BDI agents, follow the paradigm of practical reasoning, i.e. reasoning directed towards actions, by employing core modeling concepts that mimic human mental attitudes including beliefs, desires (or goals) and intentions (or plans). In this paper we show that these modeling artifacts can nicely capture the ingredients of the DEVS formalism, thus enabling directly embedding of DEVS into BDI-based agent-oriented programming frameworks.

In this paper we present our preliminary ideas about how DEVS models and their simulations can be mapped to and enacted by Jason multi-agent framework [4]. The presentation follows a simple example of a traffic light system inspired from [17]. Nevertheless, the approach is quite general, in the sense that:

i) It can be applied to significantly more complex modeling and simulation of discrete event dynamic systems;
ii) It can use other BDI-based multi-agent languages, frameworks and platforms [7];
iii) It can be extended to support other variants of DEVS, including Port-based DEVS, Parallel DEVS, Stochastic DEVS or Cellular DEVS [19].

Our research endeavor can be motivated by the following supporting elements:

i) It can expand the practical applicability of available BDI-based multi-agent frameworks for the modeling and simulation of reusable DEVS models.
ii) DEVS models mapped to BDI agents can be enhanced with cognitive features that are specific to BDI architecture, thus facilitating the modeling of more complex systems, possibly involving human and/or cognitive activities and processes.
iii) It can provide theoretical results that enable the clarification of the relationships that exist between BDI multi-agent systems and DEVS simulation models with the goal of enhancing and improving both of them.

The DEVS formalism was traditionally promoted by the control systems / automation community [21]. Nevertheless, several researchers were interested in translating DEVS models to formal specification languages specific to computer science and software engineering communities, including logics, process algebras, and timed automata [9, 12, 20]. Other researchers were interested in exploiting DEVS modularity features to enhance agent-based simulation models [1, 3, 8, 16, 22]. Finally, BDI agents were also proposed as a promising approach for enhancing simulation models of continuous and discrete dynamic systems [2, 5, 6, 15].

Nevertheless, based on our literature review, we claim that our results reported in this paper represent the first general, systematic and practical approach for translating classic DEVS simulation models to BDI-based multi-agent systems.

The paper is structured as follows. We start by providing a brief background of the DEVS modeling formalism and Jason multi-agent framework. We then outline a mapping of a simple DEVS model to Jason. The main idea is to map each DEVS module onto a distinct Jason agent capturing the simulation algorithm, as well as the hierarchical structuring of the DEVS system model. Although the underlying example system is quite simplistic, the mapping is elaborated enough containing all the necessary ingredients to allow its extension for capturing more complex DEVS models. The focus of our description provided in this paper is to present the idea of the DEVS to BDI mapping, including some details of the agents' architecture, their interconnection, as well as their message-handling plans that drive the discrete-event simulation process. The last section concludes the paper and points to future works.

2 BACKGROUND

2.1 DEVS

Note that there are many definitions and extensions of DEVS. Here we focus on classic DEVS.

Following [17], a (classic) DEVS model is composed of *atomic models*, *coupled models* and *coordinator*. The system input consists of a sequence of time-annotated events that cause the instantaneous transition of the system state. Events can be either external, i.e. generated by other models, or internal, i.e. generated by the system itself at state transitions occurring at pre-defined timeouts.

2.1.1 Atomic model.

An atomic model is defined as a tuple $\langle X, Y, S, ta, \delta_{int}, \delta_{ext}, \lambda, q_0 \rangle$ such that:

- X is the set of *input events.*
- Y is the set of *output events.*
- S is the set of *states.*
- $ta : S \rightarrow [0, +\infty)$ is the *time advance* function that defines the timeout of each state.
- $\delta_{int} : S \rightarrow S$ is the *internal transition* function.
- $\delta_{ext} : Q \times X \rightarrow S$ is the *external transition* function. Here Q designates the set of *total states* defined as $Q = \{(s, e) | s \in S, e \in [0, ta(s)]\}$.
- $\lambda : S \rightarrow Y \cup \{\phi\}$ is the *output function.* Here ϕ designates the *null or void event.*
- $q_0 = (s_0, e_0)$ such that $e_0 \in [0, ta(s_0)]$ is the *initial total state.*

At each time point the system is in a state $s \in S$. Assuming that there are no input events the system will remain in state s until timeout $ta(s)$ is elapsed (note that if $ta(s) = +\infty$ then the system will be in state s forever). When the timeout expires the system generates the output event $\lambda(s) \in Y$ or the null event ϕ and then transits to state $\delta_{int}(s)$. However, if an external event x is received at time $e \in [0, ta(s)]$ since the transition to s then the system transits to state $\delta_{ext}((s, e), x) \in S$.

2.1.2 Coupled model.

A coupled model can be recursively defined according to the "composite" pattern as being made-up of several atomic and coupled models. A coupled model contains all the ingredients required for correctly linking together its member atomic and coupled models.

A coupled model is defined as a tuple $\langle X, Y, \mathcal{D}, \mathcal{M}, \mathcal{I}, select \rangle$ with the following elements:

- X is the set of *input events* and Y is the set of *output events* of the coupled model.
- $\mathcal{D}$ is the set of *names of member (or basic) models.* Each name $d \in \mathcal{D}$ designates a member (atomic or coupled) model.
- $\mathcal{M} = \{M_d | d \in \mathcal{D}\}$ is the *set of basic models.*
- $\mathcal{I} = \{I_d \subseteq \mathcal{D} \cup \{self\} | d \in \mathcal{D} \cup \{self\}$ is the *set of influence sets.* This set defines the set of models influenced by each basic model, as well as by the coupled model itself, denoted by $self$. Direct circular references are not allowed, i.e. for each $d \in \mathcal{D} \cup \{self\}$ we have $d \notin I_d$.
- $select : 2^{\mathcal{D}} \rightarrow \mathcal{D}$ is the *tie-breaking function* that maps each subset $D \subseteq \mathcal{D}$ of conflicting basic models to a model $select(d) \in D$ that takes priority over the other models in D.

The semantics of coupled models can be defined by their mapping to atomic models through a process called "flattening" [17]. This process is quite straightforward. Firstly, the state space of the resulting model is obtained as the parallel composition (cartesian product) of the sets of total states of each basic model. Secondly, the internal and external transitions of the equivalent model are defined by identifying a single basic model inside the coupled model that actually must transit – the *imminent model*. This model is determined by firstly identifying the set of all (possible more than one) models with the minimum remaining time until a transition and then applying the *select* function in case of conflicts.

Note that we kept the definition of coupled models as simple as possible by ignoring translation functions. They allow renaming of events entering a connection before they are passed to the terminal point of the connection, essentially being tools for model reusability.

2.1.3 Simulation engine. The operational semantics of DEVS models is defined with the help of an abstract simulator that is composed of a collection of cooperating abstract simulation algorithms: atomic model abstract simulator, coupled model abstract simulator and root coordinator. These algorithms are interconnected into a tree-like structure, following the hierarchical decomposition of the system model. The simulators exchange information via message passing.

The simulation proceeds as a loop controlled by the root coordinator. Each pass through the loop contains two stages. In the first stage, information is propagated bottom-up, by letting each model to define and propagate upper in the tree the information about its next scheduled transition. By analyzing the information obtained from all the models, the root coordinator decides the next transition that will be executed, by propagating this information top-down in the tree, until the targeted model gets it and actually carries out the transition. This process continues until a termination condition is detected by the root coordinator.

2.2 BDI, AgentSpeak(L) and Jason

A multi-agent system is a computational system consisted of a collection of loosely-coupled components called software agents that interact to solve a given problem. The software agent paradigm was proposed about two decades ago to capture the new model of a "computer system situated in some environment that is capable of flexible autonomous action in order to meet its design objectives" [11].

Historically, agent-oriented programming, here understood as computer programming based on the agent paradigm, was firstly proposed more than 20 years ago as "a new programming paradigm, one based on cognitive and societal view of computation" [14].

AgentSpeak(L) is an abstract agent-oriented programming language introduced in [13]. Jason is a Java-based implementation, as well as an extension of AgentSpeak(L) [4].

AgentSpeak(L) follows the paradigm of practical reasoning, i.e. reasoning directed towards actions, and it provides an implementation of the belief-desire-intention (BDI) architecture of software agents.

According to this view an agent is a software module that (i) provides a software interface with the external world and (ii) contains three components: belief base, plan library and reasoning engine.

The agent's external world consists of the physical environment, as well as possibly other agents. Consequently, the agent interface provides three elements: sensing interface, actuation interface and communication interface. The agent uses its sensing interface to get percepts from its physical environment. The agent uses its actuation interface to perform actions on its physical environment. Finally, the agent uses its communication interface to interact by exchanging messages with other agents.

2.2.1 Belief base. It defines what an agent "knows" or "believes" about its environment at a certain time point. The BDI architecture does not impose a specific structuring of the belief base other than as a generic container of beliefs.

By default Jason uses a logical model of beliefs by structuring the belief base as a logic program composed of facts and rules. An atomic formula has the form $p(t_1, \ldots, t_n)$ such that p is a predicate symbol of arity $n \geq 0$, and t_i are logical terms for $i = 1, \ldots, n$. A belief is either a fact represented by an atomic formula a or a rule $h : -\, b_1 \,\&\, \ldots \,\&\, b_k, k \geq 1$ such that h and $b_j, j = 1, \ldots, k$ are atomic formulas.

2.2.2 Plan library. It defines the agent's "know-how" structured as a set of behavioral elements called plans. A plan follows the general pattern of event-condition-action rules and it is composed of three elements: triggering event, context and body. Formally, a plan has the form $e : c < -\, b$ where e is the triggering event, c is a plan context and b is a plan body.

A *plan body* specifies a sequence of agent activities $a_1 ; \ldots ; a_m$. Each a_i, $1 \leq i \leq m$, designates an agent activity. AgentSpeak(L) provides three types of activities: actions, goals, and belief updates. Actions define primitive tasks performed by the agent either on the environment (external actions) or internally (internal actions). Goals represent complex tasks. AgentSpeak(L) distinguishes between test goals, represented as $?a$ and achievement goals, represented as $!a$, where a is an atomic formula. Belief updates represent the assertion $+b$ or the retraction $-b$ of a belief b from the belief base.

A *plan context* is represented by a conjunction of conditions, such that each condition is either an atomic formula a or a negated atomic formula `not` a. The operator `not` is interpreted as "negation as failure" according to the standard semantics of logic programming.

A *triggering event* specifies a situation that can trigger the selection of a plan execution. An event can represent i) an assertion $+b$ or a retraction $-b$ of a belief b from the belief base or ii) an adoption $+g$ or dropping $-g$ of a goal g.

A plan will be actually selected for execution if and only if its context logically follows from the belief base. If $\mathcal{B}$ is the current belief base then this condition can be formally expressed as $\mathcal{B} \models c$.

2.2.3 Reasoning engine. Each Jason agent contains a component called "reasoning engine" or "agent interpreter" that controls the agent execution by "interpreting" the Jason code. The reasoning engine performs a reasoning cycle that consists of a fixed sequence of steps. Basically, each agent performs the following steps during the reasoning cycle: perceives the environment, updates its belief base, receives communication from other agents, selects an event, selects an applicable plan and adds it to its agenda, selects an item (called intention) for execution from the agenda, and finally executes the next step of the partially instantiated plan that represents the top of the currently selected intention.

The agent agenda is structured as a list of intentions. We can think of each intention as a stack of partially instantiated plans (somehow similar to a call stack in imperative programming) that represents an agent execution thread. So each stack represents one focus of attention of the agent. Using this approach an agent can execute concurrent activities by managing multiple focuses of attention [4].

Note that a partially instantiated plan that was created by invoking an achievement goal with $!g$ is stacked on top of the intention that invoked it. This means that this plan will be executed in the focus of attention corresponding to the invoking intention. Alternatively, an invocation with $!!g$ will create a new focus of attention for it, thus increasing the number of tasks that are executed concurrently by the agent.

3 MAPPING DEVS TO BDI USING JASON

3.1 A DEVS Example

We consider a simple traffic light system inspired from [17] that it is interruptible, i.e. it can be temporarily switched to manual mode when a policeman is pressing a button. This system can be described by a coupled model of the whole system, composed of two atomic models that describe the traffic light and the policeman.

The traffic light atomic model is described by the following elements:

- $X = \{button\_pressed\}$
- $Y = \{show\_green, show\_yellow, show\_red, show\_black\}$
- $S = \{green, yellow, red, to\_manual, to\_automatic, manual\}$
- $ta(green) = 57$, $ta(yellow) = 3$, $ta(red) = 60$, $ta(manual) = +\infty$, and $ta(to\_manual) = ta(to\_automatic) = 0$
- $\delta_{int}(green) = yellow$, $\delta_{int}(yellow) = red$, $\delta_{int}(red) = green$, $\delta_{int}(to\_manual) = manual$, and $\delta_{int}(to\_automatic) = red$
- $\delta_{ext}((s, e), button\_pressed) = to\_manual$ for all $s \in \{green, yellow, red\}$, and $e \in [0, ta(s)]$
  $\delta_{ext}((manual, e), button\_pressed) = to\_automatic$ for all $0 \leq e$
- $\lambda(green) = show\_yellow$, $\lambda(yellow) = show\_red$, $\lambda(red) = show\_green$, $\lambda(to\_manual) = show\_black$, $\lambda(to\_automatic) = show\_red$
- $q_0 = (green, 20)$

Note that *green*, *yellow* and *red* are the traffic light states in automatic mode, *to_manual* and *to_automatic* are temporary instantaneous states needed for transiting from automatic to manual mode, while *manual* describes the traffic light state in manual mode.

The traffic light outputs events when it is switching the light color in automatic mode or when it switches off light when it is transiting to manual mode.

The traffic light consumes a *button_pressed* input event that determines (i) the transition from whatever light color in automatic working mode to manual working mode, as well as (ii) the transition from the manual working mode to the red light color in automatic working mode.

The policeman is described by the following elements:

- $X = \emptyset$
- $Y = \{button\_pressed\}$
- $S = \{do\_job\}$
- $ta(do\_job) = 105$
- $\delta_{int}(do\_job) = do\_job$
- $\lambda(do\_job) = button\_pressed,$
- $q_0 = (do\_job, 8)$

The policeman has a single *do_job* state that describes his job of enabling / disabling the manual mode of the traffic light by pressing a button. This atomic DEVS component has no input events and it produces a single output event *button_pressed* when the policeman presses the button. The policeman has a periodic behavior with the period given by the timeout for pressing the button.

3.2 DEVS Example in Jason

3.2.1 Multi-agent organization of a DEVS model. Each DEVS atomic or coupled model is mapped to a Jason agent. The agent contains two parts:

- The *model definition* part that is captured using the agent's belief base. There is a specific belief base for each atomic or coupled model of the system.
- The *abstract algorithm* part that is captured by the agent's plan base. There is one plan base capturing the atomic model abstract simulator, as well as one plan base capturing the coupled model simulator. These two plan bases are reused by each agent that implements either an atomic model or a coupled model.

Separately, the root coordinator is mapped to a root agent that controls the coordination of the set of model agents. It is directly linked to the agent representing the top-level coupled (or atomic) model of the system. Recursively, each coupled model agent is directly linked downwards to each member atomic of coupled model agent.

Figure 1 captures the hierarchical organization of the agents representing the traffic light and policeman system. In particular, the figure shows that the topmost node of this hierarchy is represented by the *root* agent that coordinates the progress of the simulation. Immediately below the *root* there is the $policeman\_traffic\_light$ agent, representing a coupled DEVS model responsible for correctly linking together the component *policeman* and $traffic\_light$ agents. They represent atomic DEVS models capturing the behaviors of the policeman, as well as of the traffic light device.

Note also that the *policeman* and $traffic\_light$ agents will share the same plan base that describes the behavior of an atomic DEVS model, while the component *policeman* and $traffic\_light$ agent will include a different plan base, capturing the dynamics of a coupled DEVS model. Finally, the *root* agent will incorporate a plan base defining the behavior of the DEVS root simulation coordinator.

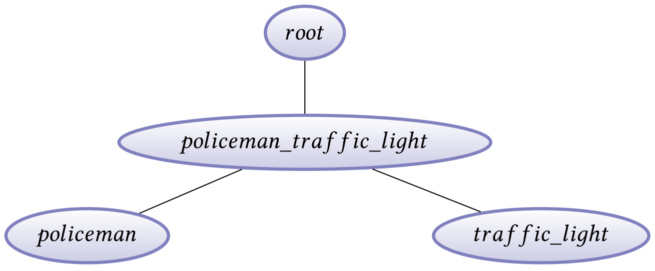

Figure 1: Traffic light and policeman system captured using Jason agents interconnected in a hierarchically structured organization

Listing 1: Belief base for configuring the *traffic_light* atomic model agent.

```
parent(policeman_traffic_light).

initial_total_state(green,20.0).

internal_transition(green,yellow).
internal_transition(yellow,red).
internal_transition(red,green).
internal_transition(to_manual,manual).
internal_transition(to_automatic,red).

external_transition(CurrentState,T,button_pressed,to_manual) :-
  .member(CurrentState,[green,yellow,red]).
external_transition(manual,T,button_pressed,to_automatic).

time_advance(green,57.0).
time_advance(yellow,3.0).
time_advance(red,60.0).
time_advance(manual,6000.0).
time_advance(to_manual,0.0).
time_advance(to_automatic,0.0).

output(green,show_yellow).
output(yellow,show_red).
output(red,show_green).
output(to_manual,show_black).
output(to_automatic,show_red).

compute_output(State,Output) :- output(State,Output).
compute_output(State,nil) :- not output(State,_).
```

3.2.2 Jason belief base of atomic and coupled models. Listing 1 presents the belief base of the *traffic_light* atomic model agent. This agent is linked up to the *policeman_traffic_light* agent that captures the representation of the top level coupled model of the system, using predicate *parent*(*ParentAgent*).

The initial total state of an atomic model is defined using predicate *initial_total_state*(*State*, *ElapsedTime*). For example, the initial state of the *traffic_light* agent is defined using the fact *initial_total_state*(*green*, 20.0) (see Listing 1).

The internal transition function of an atomic model is defined using predicate *internal_transition*(*CurrentState*, *NextState*). For example, the next state *yellow* of the current state *green* of the *traffic_light* agent is defined using fact *internal_transition*(*green*, *yellow*) (see Listing 1). Similarly, the external transition function is defined using predicate *external_transition*(*CurrentState*, *Time*, *InputEvent*, *NextState*).

The time advance function of an atomic model is defined using predicate *time_advance*(*State*, *Timeout*). For example, the time advance of state *red* of the of the *traffic_light* agent is defined using fact *time_advance*(*red*, 57.0). Note that $+\infty$ is mapped to a real number larger than the total simulation time, for example *time_advance*(*manual*, 6000.0) (see Listing 1). This specification model can be easily extended to support stochastic time advancement, as it is typically required by the modeling and simulation of stochastic systems and processes.

The output function of an atomic model is defined using predicate *output*(*State*, *OutputEvent*). For example, the output event of state *yellow* of the of the *traffic_light* agent is defined using fact *output*(*yellow*, *show_red*) (see Listing 1).

Listing 2: Belief base for configuring the *policeman_traffic_light* coupled model agent.

```
parent(root).

child(policeman).
child(traffic_light).

select([X],X).
select([policeman,traffic_light],policeman).

influences(policeman,traffic_light).
```

Listing 2 presents the belief base of the *ploceman_traffic_light* coupled model agent. Member models are recorded in the belief base as facts defined using predicate *child*(*agent*). Influence sets are defined using predicate *influences*(*Agent*, *InfluencedAgent*). Finally, conflict resolution is defined using predicate *select*(*ListOfAgents*, *Agent*).

For example, according to the facts shown in Listing 2, (i) the coupled model agent *policeman_traffic_light* has two children model agents *polceman* and *traffic_light*, (ii) agent *policeman* influences agent *traffic_light*, and (iii) agent *policeman* has higher priority than agent *traffic_light*.

3.2.3 Jason plan base of abstract simulator. The plan base of Jason agents follows the abstract simulator algorithm for atomic, respectively coupled model, presented in [17].

The plan base for the atomic model abstract simulator is used by agents *policeman* and *traffic_lights*. The plan base for the coupled model abstract simulator is defined in a similar way[1] and it is used by agent *policeman_traffic_agent*.

Note that after carrying out an internal or external transaction by plans *trans* and *input* the atomic model agent issues a *done* message with the next scheduled transition time to its parent agent.

Abstract simulator agents communicate by exchanging messages structured as *mes*(*Message*, *Agent*, *Time*). Here *Message* indicates the type and content of the message, *Agent* indicates the agent that emitted the message (with the exception of *trans* messages where it represents the target agent that must carry out the current transition), while *Time* represents a time moment with semantics dependent on the type of the message.

The abstract simulator is using several types of messages.

The *configure* message is initially emitted by atomic model agents during the initialization process, and sent up by each model (either by emitter or receiver of the message) to the parent agent. The message is used to configure initial elapsed times for each agent of the system. The *Time* argument represents the initial elapsed time, extracted from the initial total state of the atomic model. The message is processed by each coupled model to determine: i) the last transaction time of a member model, that is recorded internally in the coupled model agent belief base as a fact using predicate *last_component_time*(*Agent*, *Time*), ii) the last transaction time of the coupled model (maximum of the times recorded at previous point) that is recorded internally in the coupled model agent belief

[1]The complete Jason code that represents the policeman and traffic light system can be downloaded from http://software.ucv.ro/~cbadica/devs2bdi.zip.

base using predicate $last\_time(Time)$. The message is processed by the *root* agent to determine the moment for initiating the simulation.

The *init* message is initially emitted by the root agent to indicate the start of the simulation. The $Time$ parameter contains the initial value of the simulation time, usually equal to 0.0. Each agent (either emitter or coupled model receiver) propagates the message to its children, so the simulation initiation event is spread to all the agents of the system.

The *done* message is sent by each atomic or coupled model agent to indicate that the computation of a transition was finished for the current model. The $Time$ argument indicates the next scheduled transaction time of the model. This message is always sent to the parent agent. The message is processed by coupled model agents to record the next scheduled transaction of each member model as a fact using predicate $next\_component\_time(Agent, Time)$, as well as for updating the last transaction time of each member model and of the whole coupled model.

The *trans* message triggers a transition in the model. The message is processed by atomic model agents by carrying out the transaction. The end of this processing is signaled by the issue of a *done* message. Optionally, carrying out the transaction can issue an *output* message. The last and next transaction time of an atomic model are recorded internally in the belief base of the atomic model agent as facts composed with predicates $last\_time(Time)$ and $next\_time$. The *trans* message is also processed by coupled model agents. The processing assumes: (i) the propagation of the message to the imminent model agent (this is determined among the children of the coupled model agent), as well as (ii) the recording of the imminent model agent as an "active child" using a belief $active\_child(Agent)$.

The $output(X)$ message is emitted by atomic model agents, just before the occurrence of a timeout that generates an internal state transition. Moreover, the message is processed by coupled model agents by either (i) mapping it to an $input(X)$ message that is propagated as input to an influenced agent (in this case the $InfluencedAgent$ is recorded as "active" using a belief defined by a fact $active\_agent(InfluencedAgent)$, or (ii) by propagating it as an output of the coupled model agent itself, if the set of influenced agents contains $self$.

The $input(X)$ message is emitted when an output event is propagated to an influenced agent in a coupled model. The message is processed by atomic model agents by carrying out an external transaction. The message is also propagated to the member agents that are influenced by $self$, thus enabling its further processing by the coupled model agents.

For example, Listing 3 presents two plans used by atomic model agents to carry out internal and external transitions.

Plan *trans* has the responsibility for carrying out an internal transition of the atomic model. If the current state specifies an output event then an *output* message is sent to the parent agent, just before the internal transaction is carried out.

Plan *input* has the responsibility for processing an input received by an atomic model. The effect of receiving an input at a time moment within the time interval defined for the current state is to carry out the corresponding external transition.

Note that agent plans are using the *atomic* annotation that enforces the atomic execution of their actions. This is required for the consistent update of the state of each DEVS component.

Listing 3: Plans used by atomic model agents for carrying out internal and external transitions.

```
@trans[atomic] +mes(trans,From,Time) : next_time(Time) <-
  -mes(trans,From,Time)[source(From)];
  .print("Received: ",mes(trans,From,Time));
  ?parent(Parent);
  .my_name(This);
  ?current_state(State);
  ?compute_output(State,Output);
  .print("Time: ",Time," Output: ",Output);
  if (Output \== nil) {
    .send(Parent,tell,mes(output(Output),This,Time));
    .print("Sent: ",mes(output(Output),This,Time)," to ",Parent);
  }
  ?internal_transition(State,NextState);
  ?time_advance(NextState,TimeAdvance);
  NextTime = Time+TimeAdvance;
  -+current_state(NextState);
  -+last_time(Time);
  -+next_time(NextTime);
  .send(Parent,tell,mes(done,This,NextTime));
  .print("Sent: ",mes(done,This,NextTime)," to ",Parent).

@input[atomic] +mes(input(X),From,Time) :
  last_time(LastTime) & LastTime <= Time &
  next_time(NextTime) & Time <= NextTime <-
  -mes(input(X),From,Time)[source(From)];
  .print("Received: ",mes(input(X),From,Time));
  ?parent(Parent);
  .my_name(This);
  ElapsedTime = Time - LastTime;
  ?current_state(State);
  ?external_transition(State,ElapsedTime,X,NextState);
  ?time_advance(NextState,TimeAdvance);
  NewNextTime = Time+TimeAdvance;
  -+last_time(Time);
  -+next_time(NewNextTime);
  -+current_state(NextState);
  .send(Parent,tell,mes(done,This,NewNextTime));
  .print("Sent: ",mes(done,This,NewNextTime)," to ",Parent).
```

Finally, the *root* agent coordinates the simulation. Its belief base defines the top-level agent of the model – $policeman\_traffic\_light$, as well as the initial and final simulation times.

The definition of the *root* agent is shown in Listing 4. It is a proactive agent directed by the $!next\_event$ achievement goal. This goal is triggered by $configure$ plan, as soon as the configuration of the system was finished.

3.3 Discussion

The retrospective analysis of our implementation reveals certain advantages of using Jason agents for capturing DEVS simulation models.

Firstly, each DEVS module is naturally mapped to a Jason agent. The simulation is driven by the message-handling plans that are contained by each agent. In fact, we can see that each DEVS model can be captured using only three types of agents (atomic model agent, coupled model agent and coordinator agent), while the agents of each type share the same plan base.

Listing 4: Definition of the *root* agent that coordinates the simulation.

```
top(policeman_traffic_light).

initial_time(0.0).
final_time(3000.0).

@configure[atomic] +mes(configure,From,Time) : true <-
  !start.

@start[atomic] +!start : true <-
  .my_name(RootAgent);
  ?top(TopAgent);
  ?initial_time(InitialTime);
  .send(TopAgent,tell,mes(init,RootAgent,InitialTime));
  .print("Sent: ",mes(init,RootAgent,InitialTime)," to ",TopAgent);
  !next_event.

@next[atomic] +!next_event : top(Top) & mes(done,Top,Time) <-
  -mes(done,Top,Time)[source(Top)];
  .print("Received: ",mes(done,Top,Time));
  .my_name(RootAgent);
  ?final_time(FinalTime);
  if (Time <= FinalTime) {
    .send(Top,tell,mes(trans,RootAgent,Time));
    .print("Sent: ",mes(trans,RootAgent,Time)," to ",Top);
  }.

-!next_event : true.

@activate[atomic] +X : true <- !next_event.
```

Secondly, the hierarchical interconnection of the agents is nicely captured by the logical facts incorporated into the belief base of each agent.

Thirdly, the other elements of the DEVS model (time advance, transition and output functions) are explicitly captured as logical facts included into the belief base of each agent. Moreover, dependencies between the agents capturing basic models of a coupled DEVS model are also represented as logical facts incorporated into the belief base of coupled model agents.

Fourthly, the tie-breaking function is represented as conflict resolution logic rules that define the predicate `select/2`. Basically, the full flexibility of the Prolog interpreter that is available to a Jason agent can be reused to define very complex tie-breaking functions.

Fifthly, translation functions [17], currently not included in our models, and thus not supported by our prototype implementation, can be also easily captured using logic facts stored into the agents' belief bases.

Finally the reasoning cycle of BDI agents, controlled by the agents' message-handling plans, automatically drives the selection of events and the correct advancement of the simulation time.

Before closing our discussion is noteworthy to mention that although the Jason multi-agent platform naturally supports the parallel and distributed execution of agents, the current implementation is still based on the standard semantics of classic DEVS, thus being inherently sequential. This means that events triggering proceeds strictly in a sequential order. We plan to address this restriction in the near future by considering suitable parallel extensions of DEVS that allow the parallel triggering of events, thus naturally supporting the inherent parallelism that is present in many DEVS simulation models.

4 CONCLUSION

In this paper we proposed an approach for capturing classic DEVS simulation models using Jason agents. The approach was introduced by considering a simple DEVS model comprising two atomic models and one coupled model. Each DEVS model (either atomic or coupled) is mapped to a distinct Jason agent. The simulation is managed by a root coordinator agent that is responsible with advancing the simulation time by triggering the next scheduled event. Each coupled and atomic model agent is driven by a separate plan base. The initial results reported in this paper can be extended in many directions, including: i) modeling and simulation of more complex and realistic systems; ii) extending the approach to support other variants of DEVS (cellular, parallel, stochastic); iii) applying the approach to other BDI-based multi-agent platforms.

REFERENCES

[1] Mahuna Akplogan, Gauthier Quesnel, Frédérick Garcia, Alexandre Joannon, and Roger Martin-Clouaire. 2010. Towards a Deliberative Agent System Based on DEVS Formalism for Application in Agriculture. In *Proceedings of the 2010 Summer Computer Simulation Conference (SCSC '10)*. Society for Computer Simulation International, San Diego, CA, USA, 250–257. http://dl.acm.org/citation.cfm?id=1999416.1999447

[2] Amelia Bădică, Costin Bădică, Marius Brezovan, and Mirjana Ivanović. 2017. Simulation of Dynamic Systems Using BDI Agents: Initial Steps. In *Intelligent Distributed Computing X (Studies in Computational Intelligence)*, Costin Badica, Amal El Fallah Seghrouchni, Aurélie Beynier, David Camacho, Cédric Herpson, Koen Hindriks, and Paulo Novais (Eds.), Vol. 678. Springer International Publishing, Cham, 23–33. https://doi.org/10.1007/978-3-319-48829-5_3

[3] Paul Antoine Bisgambiglia and Romain Franceschini. 2013. Agent-oriented Approach Based on Discrete Event Systems (WIP). In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium (DEVS 13)*. Society for Computer Simulation International, San Diego, CA, USA, 27:1–27:6. http://dl.acm.org/citation.cfm?id=2499634.2499661

[4] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. 2007. *Programming Multi-Agent Systems in AgentSpeak using Jason.* John Wiley & Sons Ltd.

[5] Amelia Bădică, Costin Bădică, Mirjana Ivanović, and Daniela Dănciulescu. 2018. Multi-agent modelling and simulation of graph-based predator-prey dynamic systems: A BDI approach. *Expert Systems* (2018). https://doi.org/10.1111/exsy.12263

[6] Amelia Bădică, Costin Bădică, Florin Leon, and Ionuţ Buligiu. 2016. Modeling and Enactment of Business Agents Using Jason. In *Proceedings of the 9th Hellenic Conference on Artificial Intelligence – SETN '16*. ACM, New York, NY, USA, Article 10, 10:1–10:10 pages. https://doi.org/10.1145/2903220.2903253

[7] Costin Bădică, Zoran Budimac, Hans-Dieter Burkhard, and Mirjana Ivanović. 2011. Software agents: Languages, tools, platforms. *Computer Science and Information Systems* 8, 2 (2011), 255–298. https://doi.org/10.2298/CSIS110214013B

[8] Benjamin Camus, Christine Bourjot, and Vincent Chevrier. 2015. Combining DEVS with Multi-agent Concepts to Design and Simulate Multi-models of Complex Systems (WIP). In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (DEVS '15)*. Society for Computer Simulation International, San Diego, CA, USA, 85–90. http://dl.acm.org/citation.cfm?id=2872965.2872977

[9] Maximiliano Cristiá and Bernard P. Zeigler. 2008. Formalizing the Semantics of Modular DEVS Models with Temporal Logic. In *7e Conférence Francophone de MOdlisation et SIMulation - MOSIM'08*.

[10] Virginia Dignum, John Tranier, and Frank Dignum. 2010. Simulation of intermediation using rich cognitive agents. *Simulation Modelling Practice and Theory* 18, 10 (2010), 1526–1536. https://doi.org/10.1016/j.simpat.2010.05.011

[11] N.R. Jennings and M. Wooldridge. 1998. Applications of Intelligent Agents. In *Agent Technology*, Nicholas R. Jennings and Michael J. Wooldridge (Eds.). Springer Berlin Heidelberg, 3–28.

[12] Jean-Pierre Müller. 2009. Towards a Formal Semantics of Event-Based Multi-agent Simulations. In *Multi-Agent-Based Simulation IX*, Nuno David and Jaime Simão Sichman (Eds.). Lecture Notes in Computer Science, Vol. 5269. Springer Berlin Heidelberg, Berlin, Heidelberg, 110–126. https://doi.org/10.1007/978-3-642-01991-3_9

[13] Anand S. Rao. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, Walter Van de Velde and J. W. Perram (Eds.). Lecture Notes in Computer Science, Vol. 1038. Springer, 42–55. https://doi.org/10.1007/BFb0031845
[14] Y. Shoham. 1993. Agent-oriented Programming. *Artificial Intelligence* 60, 11 (1993), 51–92.
[15] Dhirendra Singh, Lin Padgham, and Brian Logan. 2016. Integrating BDI Agents with Agent-Based Simulation Platforms. *Autonomous Agents and Multi-Agent Systems* 30, 6 (2016), 1050–1071. https://doi.org/10.1007/s10458-016-9332-x
[16] Alexander Steiniger, Frank Krüger, and Adelinde M. Uhrmacher. 2012. Modeling Agents and Their Environment in multi-level-DEVS. In *Proceedings of the Winter Simulation Conference (WSC '12)*. Winter Simulation Conference, 233:1–233:12. https://doi.org/10.1109/WSC.2012.6465113
[17] Yentl Van Tendeloo and Hans Vangheluwe. 2017. An Introduction to Classic DEVS. *ArXiv e-prints* (Jan. 2017). arXiv:cs.OH/1701.07697
[18] Yentl Van Tendeloo and Hans Vangheluwe. 2017. An Evaluation of DEVS Simulation Tools. *Simulation* 93, 2 (Feb. 2017), 103–121. https://doi.org/10.1177/0037549716678330
[19] Gabriel A. Wainer. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach* (1st ed.). CRC Press, Inc., Boca Raton, FL, USA.
[20] Aznam Yacoub, Maamar el Amine Hamri, and Claudia Frydman. 2017. Restricting DEv-PROMELA with a Hierarchy of Simulation Formalisms. In *Proceedings of the Symposium on Theory of Modeling & Simulation – TMS/DEVS '17*. Society for Computer Simulation International, San Diego, CA, USA, 13:1–13:11. http://dl.acm.org/citation.cfm?id=3108905.3108918
[21] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. 2000. *Theory of Modeling and Simulation, Second Edition.* Academic Press.
[22] Mingxin Zhang, Mamadou Seck, and Alexander Verbraeck. 2013. A DEVS-based M&S Method for Large-scale Multi-agent Systems. In *Proceedings of the 2013 Summer Computer Simulation Conference (SCSC '13)*. Society for Modeling & Simulation International, Vista, CA, 3:1–3:8. http://dl.acm.org/citation.cfm?id=2557696.2557700