Modeling Dynamic Domains with ConGolog

Yves Lespérance,
Dept. of Computer Science, York University,
Toronto, ON Canada, M3J 1P3
lesperan@cs.yorku.ca

Todd G. Kelley, John Mylopoulos, and Eric S.K. Yu
Dept. of Computer Science, University of Toronto,
Toronto, ON Canada, M5S 1A4
{tgk,jm,eric}@cs.toronto.edu

Abstract

In this paper, we describe the process specification language *ConGolog* and show how it can be used to model business processes for requirements analysis. In *ConGolog*, the effects of actions in a dynamic domain are specified in a logical framework. This supports modeling even in the absence of complete information. The behavior of agents in the domain is specified in a concurrent process language, whose semantics is defined in the same logical framework. We then describe a simulation tool implemented in terms of logic programming technology. As well, we discuss a verification tool which is being developed based on theorem proving technology.

1 Introduction

Dynamic models of aspects of the world constitute an essential ingredient of information systems engineering. Such models are useful during requirements analysis where the operational environment of a system-to-be needs to be described, along with the role the system will play within that environment. Dynamic models also play a key role during design when the functions of the system and its major components are specified.

Existing dynamic models come in two flavors. State-based models describe the processes of a dynamic world in terms of states and (state) transitions. Finite-state machines, Petri nets [14], statecharts [7], and workflows are examples of modeling frameworks which adopt a state-oriented view of the world. A major advantage of state-based models is that they can be simulated, showing the sequence of state transitions that will take place for a particular sequence of input signals. Alternatively, predicative models [1, 20] describe processes in terms of pre/post-conditions, i.e., in terms of a condition that has to be true before a process is launched (the precondition) and a condition that will be true once the process execution has been completed (the postcondition). Predicative models admit a type of formal analysis where properties of a process can be verified. For example, one can show that a certain invariant is preserved by a process in the sense that if the invariant holds before the process begins, it will also hold at the end of the process.

Predicative models typically do not support simulation and state-based models do not support formal property analysis. This paper describes a modeling framework for dynamic worlds that supports both simulation and verification. The framework is based on the language *ConGolog*, originally developed as a high level language for programming robots and software agents [3].[1] *ConGolog* is based on a logical formalism, the situation calculus, and can model multi-agent processes, nondeterminism, as well as concurrency. Because of its logical foundations, *ConGolog* can accommodate incompletely specified models, either in the sense that the initial state of the system is not completely specified, or in the sense that the processes involved are nondeterministic and may evolve in any number of ways. These features are especially useful when one models business processes and open-ended real world situations.

Section 2 of the paper introduces the framework and how it is used for modeling states, actions, and processes, and presents a simple example involving the handling of orders by a business. Section 3 demonstrates the ability of the framework to support both simulation and verification, while section 4 describes the semantics of *ConGolog* and the formal theory on which it is based. Finally, section 5 summarizes the contributions of this research and suggests directions for further research.

2 Modeling a Domain in *ConGolog*

In the *ConGolog* framework, an application domain is modeled *logically* so as to support reasoning about the specification. A *ConGolog* model of a domain involves two components. The first component is a specification of the *domain dynamics*, i.e. how the states are modeled, what actions may be performed, when they are possible, what their effects are, and what is known about the initial state of the system. This component is specified in a purely declarative way, in a language called the *Golog*

[1] *ConGolog* is an extended version of the Golog (AlGOl in LOGic) language described in [10]. Earlier work on modeling business processes in Golog appeared in [15].

Domain Language (GDL).[2]

The second component of a *ConGolog* domain model is a specification of the *processes* that are unfolding in the domain; this can also be viewed as a specification of the behavior of the agents in the domain. Because we are interested in modeling domains involving complex processes, this component is specified procedurally in the *ConGolog* process description language.

Both GDL and the *ConGolog* process language have formal semantics defined in a language of predicate logic called the *situation calculus*. Various mechanisms for reasoning about properties of a domain have been implemented using this situation calculus semantics. We outline the semantics in section 4.

To illustrate the use of the framework to model a domain, we use a running example involving a simple mail-order business. We assume that the business sells only one product. We also assume that there are only two agents in the business, who could be single people or whole departments:

- the *order desk operator*, who processes payment for orders while waiting for the phone to ring, and when it does, receives an order from a customer; and
- the *warehouse operator*, who fills the orders that the order desk operator has received, and ships orders for which the order desk operator has processed payment; whenever a shipment is delivered by a supplier, the warehouse operator receives the shipment.

Orders can be processed in two possible ways as described in the diagram in Figure 1. The example is kept artificially simple so that it can be presented in its entirety.

2.1 Modeling Domain Dynamics in GDL

The first component of a *ConGolog* model is a specification of the dynamics of the domain and of what is known about its initial state. For this, our framework uses GDL. In our models, we imagine the world as starting out in a particular initial *situation* (or state), and evolving into various other possible situations through the performance of *actions* by various agents. Situations are described in terms of *fluents*. *Relational fluents* are relations or properties whose truth value can vary from situation and to situation and *functional fluents* are functions whose value varies from situation to situation. For instance in our example, we use the action term $shipOrder(agt, order)$ to represent the action of agent agt shipping $order$, and the relational fluent $OrderShipped(order)$ to represent the property that $order$ has

[2]GDL is related to Gelfond and Lifschitz's action language $\mathcal{A}$ [6]; one significant difference though, is that GDL is a first-order language, while $\mathcal{A}$ is essentially a propositional language.

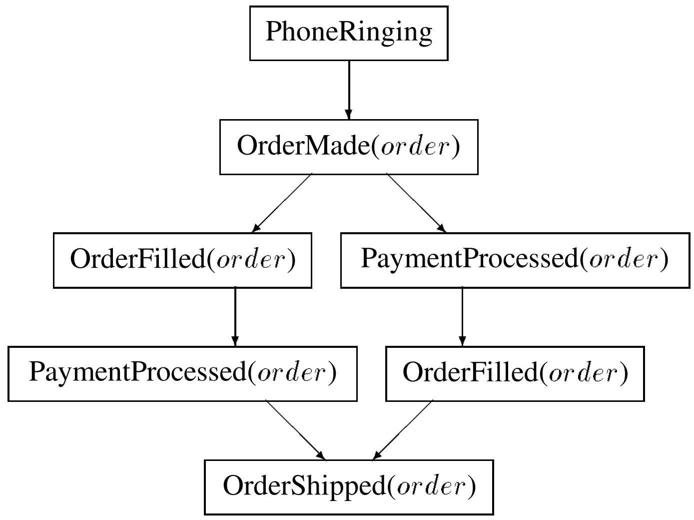

Figure 1: An informal diagram showing the two possible paths in the life-cycle of an order.

been shipped. This fluent might be false in the initial situation, but true in a situation that is the result of the action $shipOrder(agt, order)$.

The modeler chooses the fluents and actions in a domain model according to the desired level of abstraction. A GDL domain specification starts with a set of declarations for the fluents used in the model. Each *fluent declaration* specifies the name of the fluent, the number of arguments it takes and whether it is a functional fluent or not. Optionally, one can also specify what value the fluent has in the initial situation. The GDL fluent declarations for our example domain appear in Figure 2. Note that orders are identified by a number determined by the value of the $orderCounter$ fluent when the order is received.

Next, a GDL domain specification includes *action declarations*, one for each primitive action. These specify the name of the action, the arguments it takes, and action's preconditions, i.e. the conditions under which it is possible. The GDL action declarations for our example domain appear in Figure 2. The last two represent actions performed by customers and suppliers that impact on the business. We view customers and suppliers as agents that are outside the system and are not interested in modeling their behavior in detail. We only consider their effect on the system through these two actions, which we call *exogenous* actions.

Finally, a GDL specification includes a set of *effect declarations*, one for each fluent that is affected by an action. The effect declarations for our example domain

Fluent Declarations

fluent $PhoneRinging()$	% the phone is ringing
initially $False$;	
fluent $OrderMade(order)$	% $order$ has been made
initially $False$;	
fluent $PaymentProcessed(order)$	% payment for $order$ has been
initially $False$;	% processed
fluent $OrderFilled(order)$	% $order$ has been filled
initially $False$;	
fluent $OrderShipped(order)$	% $order$ has been shipped
initially $False$;	
fluent $SuppliesAtShippingDock()$	% incoming supplies are at
initially $False$;	% the shipping dock
functional fluent $orderQuantity(order)$	% the quantity of items requested
initially 0;	% in $order$
functional fluent $orderCounter()$	% the value of the order counter
initially 1;	
functional fluent $stock()$	% the quantity of items in stock
initially 10;	
functional fluent $incomingOrderQuantity()$	% the quantity of items requested
initially 0;	% in the incoming order
functional fluent $incomingSuppliesQuantity()$	% the quantity of items delivered
initially 0;	% in the incoming shipment

Action Declarations

action $receiveOrder(agt)$	% agt receives the incoming phone order
possible when $PhoneRinging()$;	
action $processPayment(agt, order)$	% agt processes payment for $order$
possible when $OrderMade(order)$;	
action $fillOrder(agt, order)$	% agt fills $order$
possible when $OrderMade(order)$;	
action $shipOrder(agt, order)$	% agt ships $order$
possible when $OrderMade(order) \land OrderFilled(order)$;	
action $receiveSupplies(agt)$	% agt receives supplies at the loading dock
possible when $SuppliesAtShippingDock()$	
$\land\ stock() + incomingSuppliesQuantity() < 100$;	
exogenous action $mkOrder(cust, q)$	% customer $cust$ makes an order for q items
possible when $\neg PhoneRinging()$;	
exogenous action $deliverSupplies(supp, q)$	% supplier $supp$ delivers q items of new stock
possible when $\neg SuppliesAtShippingDock()$;	

Figure 2: Example GDL domain specification – part 1.

Effect Declarations
occurrence $receiveOrder(agt)$ **results in** $OrderMade(orderCounter())$ **always**;
occurrence $receiveOrder(agt)$ **results in**
$orderCounter() = orderCounter() + 1$ **always**;
occurrence $receiveOrder(agt)$ **results in** $\neg PhoneRinging()$ **always**;
occurrence $receiveOrder(agt)$ **results in**
$orderQuantity(orderCounter()) = incomingOrderQuantity()$ **always**;
occurrence $processPayment(agt, order)$ **results in** $PaymentProcessed(order)$ **always**;
occurrence $receiveSupplies(agt)$ **results in** $\neg SuppliesAtShippingDock()$ **always**;
occurrence $receiveSupplies(agt)$ **results in**
$stock() = stock() + incomingSuppliesQuantity()$ **always**;
occurrence $fillOrder(agt, order)$ **results in** $OrderFilled(order)$
when $orderQuantity(order) < stock()$;
occurrence $fillOrder(agt, order)$ **results in** $stock() = stock() - orderQuantity(order)$
when $orderQuantity(order) < stock()$;
occurrence $shipOrder(agt, order)$ **results in** $OrderShipped(order)$ **always**;
occurrence $mkOrder(c, q)$ **results in** $PhoneRinging()$ **always**;
occurrence $mkOrder(c, q)$ **results in** $incomingOrderQuantity() = q$ **always**;
occurrence $deliverSupplies(su, q)$ **results in** $SuppliesAtShippingDock()$ **always**;
occurrence $deliverSupplies(su, q)$ **results in** $incomingSuppliesQuantity() = q$ **always**;

Figure 3: Example GDL domain specification – part 2.

appear in Figure 3. Note the declaration for the action $fillOrder$. Its effects depend on the context, in that it only causes the order to become filled when there is sufficient stock to do so; otherwise, the action behaves as a no-op.

2.2 Modeling Domain Processes in *ConGolog*

As mentioned earlier, a *ConGolog* domain model includes a second component that describes the processes unfolding in the domain. This is specified in a procedural sublanguage where actions can be composed into complex processes, possibly involving concurrency and nondeterminism. This *ConGolog* process specification language provides the constructs listed in Figure 4.

Let us go over some of the less familiar constructs in the language. The nondeterministic constructs include $(\sigma_1 \mid \sigma_2)$, which nondeterministically choses between processes σ_1 and σ_2, $\pi\vec{x}[\sigma]$, which nondeterministically picks a binding for the variables in the list $\vec{x}$ and performs the process σ for this binding of $\vec{x}$, and σ^*, which means performing σ zero or more times. Concurrent processes are modeled as interleavings of the actions involved. The actions themselves are viewed as atomic and cannot be interrupted. A process may become blocked when it reaches a

α	primitive action
$\phi?$	wait for a condition
$(\sigma_1; \sigma_2)$	sequence
$(\sigma_1 \mid \sigma_2)$	nondeterministic choice between actions
$\pi\, \vec{x}\, [\sigma]$	nondeterministic choice of arguments
σ^*	nondeterministic iteration
if ϕ **then** σ_1 **else** σ_2 **endIf**	conditional
while ϕ **do** σ **endWhile**	loop
$(\sigma_1 \parallel \sigma_2)$	concurrent execution
$(\sigma_1 \rangle\!\rangle\, \sigma_2)$	concurrency with different priorities
$\sigma^{\parallel}$	concurrent iteration
$<\vec{x} : \phi \rightarrow \sigma>$	interrupt
proc $\beta(\vec{x})$ σ **endProc**	procedure definition
$\beta(\vec{t})$	procedure call
noOp	do nothing

Figure 4: Constructs in *ConGolog* process specifications.

primitive action whose preconditions are false or a wait action $\phi?$ whose condition ϕ is false. Then, execution of the system may continue provided another process executes next. In $(\sigma_1 \rangle\!\rangle\, \sigma_2)$, σ_1 has higher priority than σ_2, and σ_2 may only execute when σ_1 is done or blocked. $\sigma^{\parallel}$ is like nondeterministic iteration σ^*, but the instances of σ are executed concurrently rather than in sequence. Finally, an interrupt $<\vec{x} : \phi \rightarrow \sigma>$ has a list of variables $\vec{x}$, a trigger condition ϕ, and a body σ. If the interrupt gets control from higher priority processes and the condition ϕ is true for some binding of the variables, the interrupt triggers and the body is executed with the variables taking these values. Once the body completes execution, the interrupt may trigger again. With interrupts, it is easy to write process specifications that are reactive in that they will suspend whatever task they are doing to handle given conditions as they arise.

Let us look at how this language can be used to specify the processes in our mail-order business domain; the specification appears in Figure 5. The whole system is specified by the $main$ procedure. It executes two concurrent processes, one for each agent in the domain. The agents in our system are very reactive. Their behavior involves monitoring the progress of orders, and when certain conditions hold, performing some step in the processing of the order. So we specify their behavior using interrupts. The behavior of the order desk operator is specified by the $runOrderDesk$ procedure. This agent has two responsibilities: receiving an order when the phone rings and processing payments for orders. Each of these is han-

Process Specifications

proc $runOrderDesk(odAgt)$
$< phoneRinging \rightarrow receiveOrder(odAgt) >$
$\rangle\rangle$
$< order : OrderMade(order) \wedge \neg paymentProcessed(order)$
$\rightarrow processPayment(odAgt, order) >$
endProc

proc $runWarehouse(wAgt)$
$< SuppliesAtShippingDock \rightarrow receiveSupplies(wAgt) >$
$\rangle\rangle$
$< order : OrderMade(order) \wedge OrderFilled(order)$
$\wedge PaymentProcessed(order) \wedge \neg OrderShipped(order)$
$\rightarrow shipOrder(wAgt) >$
$\rangle\rangle$
$< order : OrderMade(order) \wedge \neg OrderFilled(order)$
$\rightarrow fillOrder(wAgt) >$
endProc

proc $main$
$runOrderDesk(OrderDeskAgt) \parallel runWarehouse(WarehouseAgt)$
endProc

Figure 5: Example *ConGolog* process specification.

dled by an interrupt. Since receiving orders when the phone rings is more urgent than processing payments, the interrupt for receiving orders runs at higher priority than the one for processing payments. The interrupt for processing payment non-deterministically picks an order for which payment has not yet been processed, and processes its payment.

The $runWarehouse$ procedure specifying the behavior of the warehouse operator involves three interrupts each running at a different priority. At the highest priority, the operator should receive an incoming shipment when the shipping door bell rings. When there is no shipment to receive, the next highest priority is to ship orders that are ready to ship (orders that are filled and for which payment is processed), if there are any, picking at random. At the lowest priority, the operator should fill any order that has been received but not yet filled, picking the order arbitrarily.[3]

3 Analyzing Domain Specifications using *ConGolog* Tools

3.1 Validation through Simulation

Simulation is a useful method for validating domain models. We have developed a tool for incrementally generating execution traces of *ConGolog* process specifications. This tool can be used to check whether a model executes as expected in various conditions. For example, our simulation tool can be used to confirm that our model of the mail-order business domain can process a single order in two different ways, either filling the order before payment is processed, or vice versa. In Figure 6, we see a trace of the first execution of the specification, where a single order by customer $c1$ for 3 items $(mkOrder(c1, 3))$ is made and where payment is processed on the order before it is filled. The figure shows the simulation tool at the end of the execution. A list of executed actions appears at the top right of the viewer, with later action occurrences at the top. The (partial) state of the system is displayed at the top left of the viewer. A trace of a second execution of the specification where the order is filled before payment is processed appears in Figure 7.

Our simulation tool is based on a logic programming technology implementation of the *ConGolog* framework. It involves two main components:

- the *GDL compiler*, which takes a domain specification in GDL and produces

[3]Note that the action $fillOrder$ succeeds in filling the order only if there is sufficient stock, otherwise it has no effect. Thus, the agent may repeatedly attempt to fill an order until it succeeds. If one wants to ensure that the agent gives priority to the orders that it has sufficient stock to fill, one needs to add a copy of the interrupt with the additional condition $orderQuantity(order) < stock$ running at a higher priority than the existing interrupt.

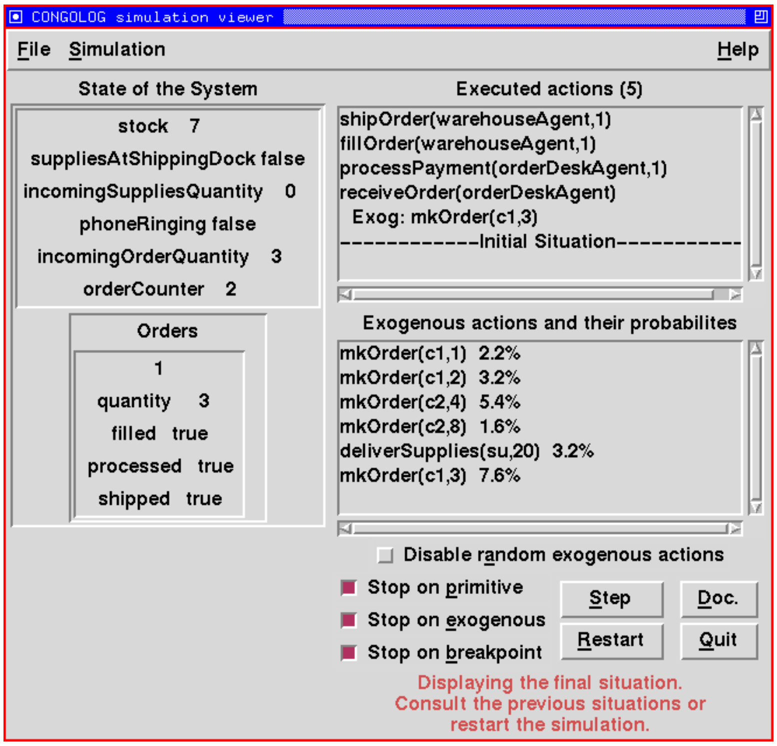

Figure 6: The *ConGolog* simulation tool at the end of a first execution of the mail-order domain specification.

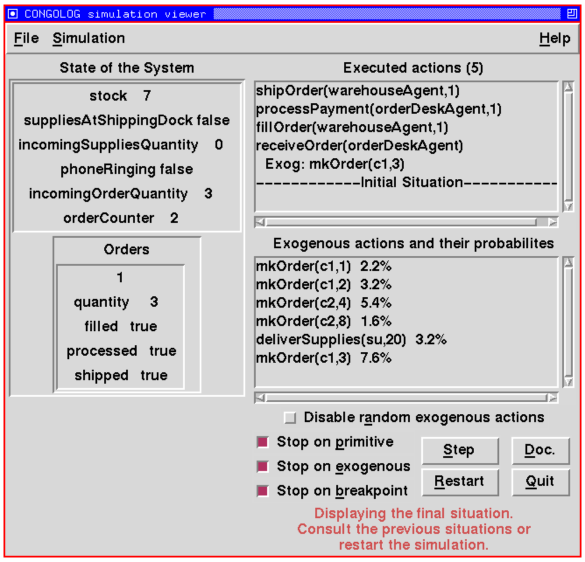

Figure 7: The simulation tool at the end of a second execution of the mail-order domain specification.

a Prolog implementation of the corresponding situation calculus domain theory;

- the ConGolog *interpreter*, which takes a *ConGolog* process specification and a domain theory, and generates execution traces that satisfy the process specification given the domain theory; the interpreter uses the domain theory in evaluating tests and checking whether action preconditions are satisfied as it generates the execution traces; the Prolog implementation of the interpreter is described in [4].

These two components are at the core of a toolkit. The kit includes a *graphical viewer*, shown in Figures 6 and 7 for displaying simulations of *ConGolog* process specifications. This tool, which is implemented in Tcl/Tk, displays the sequence of actions performed by the *ConGolog* process specification and the value of the fluents in the resulting situation (or any situation along the path). The process execution can be stepped through and exogenous events can be generated either manually or at random according to a given probability distribution. The manner in which state information is displayed can be specified easily and customized as required.

The toolkit also includes a module for *progressing* the initial situation, i.e. updating the specification of the initial situation to make it correspond to a later situation [12]. This makes the reasoning performed by the system more efficient and allows it to simulate the execution of long running processes.

The logic programming technology implementation of the *ConGolog* framework is fairly efficient and can be used for both simulation and for deploying actual applications when one provides implementations for the actions used. However, the current implementation is limited to specifications of the initial situation that can be represented as logic programs, which are essentially closed-world theories. This is a limitation of the logic programming implementation, not the *ConGolog* framework. Note also that we are currently working on extending the implementation to support limited types of incompleteness.

3.2 Verification

One may be interested in verifying that the processes in a domain satisfy certain properties. The *ConGolog* framework supports this through its logic-based semantics. For example, given our specification of the mail-order business domain, we may be interested in showing that no order is ever shipped before payment is processed, i.e.

$$\forall order.\, OrderShipped(order) \supset PaymentProcessed(order).$$

In fact, we can prove that if the above property holds in the initial situation, it will hold for every situation during an execution of our process specification. Intuitively, this is the case because (1) once payment is processed for an order no action can cause it to become unprocessed, (2) the only action that can cause an order to have been shipped is $shipOrder$, and (3) in the process specified, $shipOrder$ is only performed when payment has been processed on the order. We give a proof of the property in section 4.4. Note that the property follows even if the domain specification includes no information about the initial situation other than the fact that the property holds initially.

Another important property to verify for a mail-order business is that it should have income, i.e. that there is a situation where

$$\exists order\ PaymentProcessed(order).$$

holds. However, for the process specifications given earlier there is no guarantee of this. Hence, the business is not certain to make any money, even if it gets many orders. In fact, we can prove that if the phone always rings as soon as any order is taken, no payment would ever be processed, because the order desk operator would be too busy answering the phone.

This problem can be fixed by introducing the concept of a backlog of orders with the following declarations

functional fluent $backLog()$ **initially** 1;
occurrence $receiveOrder(agt)$ **results in**
$backLog() = backLog() + 1$ **always**;
occurrence $processPayment(agt, order)$ **results in**
$backLog() = backLog() - 1$ **always**;

and changing the highest priority interrupt of the $runOrderDesk$ procedure from

$$< PhoneRinging \to receiveOrder(odAgt) >$$

to

$$< PhoneRinging \wedge backLog() < 10 \to receiveOrder(odAgt) >$$

With these changes to the specification, one can verify that if at least one order is placed and there is sufficient stock to fill it, then the business has some income (note that a steady stream of supplies deliveries cannot cause problems because the warehouse is assumed to have a fixed capacity; see the action declaration for $receiveSupplies$).

A user-assisted verification tool that can handle arbitrary *ConGolog* theories, i.e. incompletely specified initial situations and specifications of agents' mental states (knowledge and goals), is being developed [19]. The user would provide a proof strategy and the tool would produce the detailed steps of a proof automatically. The tool is based on theorem proving technology and relies on an encoding of the *ConGolog* semantics in a form that the PVS program verification system can reason with.

4 *ConGolog* Semantics

In this section, we describe the logical foundations of the *ConGolog* framework. These foundations are what supports its use in both simulation and verification.

4.1 The Situation Calculus and the Semantics of GDL

As mentioned earlier, the semantics of GDL and of the *ConGolog* process specification language are specified in the *situation calculus* [13], a language of predicate logic for representing dynamic domains. The reasoning performed by our tools is also based on the situation calculus. Let us briefly introduce this language. In the situation calculus, all changes to the world are the result of named *actions*. A possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant S_0 is used to denote the initial situation, namely that situation in which no actions have yet occurred. There is a distinguished binary function symbol do and the term $do(\alpha, s)$ denotes the situation resulting from action α being performed in situation s. Actions may be parameterized. So for example, $do(receiveOrder(agt), s)$ would denote that situation resulting from agt having received the incoming phone order when the world was in situation s. Notice that in the situation calculus, actions are denoted by function symbols, and situations (world histories) are also first order terms. For example,

$$do(receiveOrder(OrderDeskAgent), do(mkOrder(Customer1, 2), S_0))$$

is a situation denoting the world history consisting of the sequence of actions

$$[mkOrder(Customer1, 2), receiveOrder(OrderDeskAgent)].$$

In the situation calculus, relational fluents are represented by predicate symbols that take a situation term as their last argument. This makes the dependence of the value of the fluent on the situation explicit. So for example, the formula $\neg\exists order\, OrderMade(order, S_0)$ would be used to represent the fact no order has been made in the initial situation, and

$$\begin{aligned}&OrderMade(1, do(receiveOrder(OrderDeskAgent),\\&\qquad do(mkOrder(Customer1, 2), S_0)))\end{aligned}$$

would be used to represent the fact that order number 1 has been made in the situation obtained after customer $Customer1$ makes an order for 2 items and the order desk agent receives it. Similarly, functional fluents are represented by function symbols that take a situation as their last argument, as in $orderQuantity(order, s)$, i.e., the quantity of items requested in $order$ in situation s. In GDL specifications, the situation argument of fluents is suppressed to make the notation less verbose.

The semantics of GDL maps GDL declarations into situation calculus axioms that capture the meaning of the declarations. GDL fluent declarations can include information about the value of the fluent in the initial situation in an **initially** clause. If such a clause is present, it is mapped into an *initial situation axiom*. For our mail-order business example, the fluent declarations in Figure 2 are mapped into the following initial situation axioms:

Initial Situation Axioms

$$\begin{aligned}&\neg PhoneRinging(S_0)\\&\neg OrderMade(order, S_0)\\&\neg PaymentProcessed(order, S_0)\\&\neg OrderFilled(order, S_0)\\&\neg OrderShipped(order, S_0)\\&\neg SuppliesAtShippingDock(S_0)\end{aligned}$$

$$
\begin{aligned}
&orderQuantity(order, S_0) = 0 \\
&orderCounter(S_0) = 1 \\
&stock(S_0) = 10 \\
&incomingOrderQuantity(S_0) = 0 \\
&incomingSuppliesQuantity(S_0) = 0
\end{aligned}
$$

A GDL action declaration specifies the preconditions of the action, i.e. the conditions under which it is physically possible to perform it. Such a declaration is mapped by the GDL semantics into an *action precondition axiom*. These axioms use the special predicate $Poss$, with $Poss(\alpha, s)$ representing the fact that action α is physically possible (i.e. executable) in situation s. For our example domain, the action declarations in Figure 2 are mapped into the following action precondition axioms are:

Action Precondition Axioms

$$Poss(receiveOrder(agt), s) \equiv PhoneRinging(s)$$

$$Poss(processPayment(agt, order), s) \equiv OrderMade(order, s)$$

$$\begin{aligned}&Poss(receiveSupplies(agt), s) \equiv SuppliesAtShippingDock(s) \\ &\quad \wedge stock() + incomingSuppliesQuantity() < 100\end{aligned}$$

$$Poss(fillOrder(agt, order), s) \equiv OrderMade(order, s)$$

$$\begin{aligned}&Poss(shipOrder(agt, order), s) \equiv \\ &\quad OrderMade(order, s) \wedge OrderFilled(order, s)\end{aligned}$$

$$Poss(mkOrder(customer, quantity), s) \equiv \neg PhoneRinging(s)$$

$$\begin{aligned}&Poss(deliverSupplies(supplier, quantity), s) \equiv \\ &\quad \neg SuppliesAtShippingDock(s)\end{aligned}$$

Finally, we also have GDL effect declarations which specify how actions affect the state of the world. These declarations are mapped by the GDL semantics into *effect axioms*. Effect axioms provide the "causal laws" for the domain of application. For our example, the effect declarations that appear in Figure 3 are mapped into the following effect axioms:

Effect Axioms

$$OrderMade(do(receiveOrder(agt), s))$$

$$orderCounter(do(receiveOrder(agt), s)) = orderCounter(s) + 1$$

$$\neg PhoneRinging(do(receiveOrder(agt), s))$$

$$\begin{aligned}&orderQuantity(orderCounter(s), do(receiveOrder(agt), s)) = \\ &\quad incomingOrderQuantity(s)\end{aligned}$$

$PaymentProcessed(order, do(processPayment(agt, order), s))$

$\neg SuppliesAtShippingDock(do(receiveSupplies(agt), s))$

$stock(do(receiveSupplies(agt), s)) = stock(s) + incomingSuppliesQuantity(s)$

$orderQuantity(order, s) < stock(s) \supset OrderFilled(do(fillOrder(agt, order), s))$

$$orderQuantity(order, s) < stock(s) \supset \\ \quad stock(do(fillOrder(agt, order), s)) = stock(s) - orderQuantity(order)$$

$OrderShipped(do(shipOrder(agt, order), s))$

$PhoneRinging(do(mkOrder(customer, quantity), s))$

$incomingOrderQuantity(do(mkOrder(customer, q), s)) = q$

$SuppliesAtShippingDock(do(deliverSupplies(supplier, quantity), s))$

$incomingSuppliesQuantity(do(deliverSupplies(supplier, q), s)) = q$

The full syntax and semantics of GDL are defined in [9].

4.2 Addressing the Frame Problem

The sort of logic-based framework we have described allows very incomplete information about a dynamic domain to be specified. But this creates difficulties in reasoning about action and change Effect axioms state what must change when an action is performed, but do not specify what aspects of the domain remain unchanged. One way to address this is to add *frame axioms* that specify when fluents remain unchanged by actions. For example, an agent agt filling $order$ does not cause new supplies to appear at the shipping dock:

$$\neg SuppliesAtShippingDock(s) \supset \\ \neg SuppliesAtShippingDock(do(fillOrder(agt, order), s))$$

The frame problem arises because the number of these frame axioms is very large, in general, of the order of $2 \times \mathcal{A} \times \mathcal{F}$, where $\mathcal{A}$ is the number of actions and $\mathcal{F}$ the number of fluents. This complicates the task of axiomatizing a domain and can make automated reasoning extremely inefficient. Most predicative approaches do not address this problem [2].

To deal the frame problem, we use an approach due to Reiter [17]. The basic idea behind this is to collect all effect axioms about a given fluent and make a *completeness assumption*, i.e. assume that they specify all of the ways that the value of the fluent may change. A syntactic transformation can then be applied to obtain a *successor state axiom* for the fluent, for example:

$$\begin{aligned}&PhoneRinging(do(a,s)) \equiv\\ &\quad \exists customer, quantity\ a = mkOrder(customer, quantity)\\ &\quad \vee PhoneRinging(s) \wedge \neg\exists agt\ a = receiveOrder(agt)\end{aligned}$$

This says that the phone is ringing in the situation resulting from action a being performed in situation s if and only if a is some customer making an order or if the phone was already ringing in situation s and a is not some agent receiving the order. Therefore, no other action has any effect on $PhoneRinging$. This approach yields a solution to the frame problem — a parsimonious representation for the effects of actions. Note that it relies on quantification over actions.[4]

For our example domain, applying the method to the other fluents yields the following axioms:

Successor State Axioms

$$\begin{aligned}&OrderMade(order, do(a,s)) \equiv\\ &\quad a = receiveOrder(agt) \wedge order = orderCounter(s)\\ &\quad \vee OrderMade(order, s)\end{aligned}$$

$$\begin{aligned}&PaymentProcessed(order, do(a,s)) \equiv\\ &\quad \exists agt\ a = processPayment(agt, order)\\ &\quad \vee PaymentProcessed(order, s)\end{aligned}$$

$$\begin{aligned}&OrderFilled(order, do(a,s)) \equiv\\ &\quad \exists agt\ a = fillOrder(agt, order) \vee OrderFilled(order, s)\end{aligned}$$

$$\begin{aligned}&OrderShipped(order, do(a,s)) \equiv\\ &\quad \exists agt\ a = shipOrder(agt, order) \vee OrderShipped(order, s)\end{aligned}$$

$$\begin{aligned}&SuppliesAtShippingDock(do(a,s)) \equiv\\ &\quad \exists supplier, quantity\ a = deliverSupplies(supplier, quantity)\\ &\quad \vee SuppliesAtShippingDock(s) \wedge \neg\exists agt\ a = receiveSupplies(agt)\end{aligned}$$

$$\begin{aligned}&orderQuantity(order, do(a,s)) = q \equiv\\ &\quad \exists agt\ a = receiveOrder(agt)\\ &\qquad \wedge order = orderCounter(s) \wedge q = incomingOrderQuantity(s)\\ &\quad \vee q = orderQuantity(order, s)\end{aligned}$$

$$\begin{aligned}&orderCounter(do(a,s)) = n \equiv\\ &\quad \exists agt\ a = receiveOrder(agt) \wedge n = orderCounter(s) + 1\\ &\quad \vee n = orderCounter(s)\end{aligned}$$

[4] This discussion assumes that there are no state constraints; a treatment for these that is compatible with the above approach is presented in [11].

$$
\begin{aligned}
&stock(do(a, s)) = q \equiv \\
&\quad \exists agt\, a = receiveSupplies(agt) \wedge \\
&\qquad q = stock(s) + incomingSuppliesQuantity(s) \\
&\quad \vee \exists agt, order (a = fillOrder(agt, order) \wedge \\
&\qquad q = stock(s) - orderQuantity(order)) \\
&\quad \vee q = stock(s) \wedge \neg\exists agt\, a = receiveSupplies(agt) \wedge \\
&\qquad \neg\exists agt, order\, a = fillOrder(agt, order)
\end{aligned}
$$

$$
\begin{aligned}
&incomingOrderQuantity(do(a, s)) = q \equiv \\
&\quad \exists customer\, a = mkOrder(customer, q) \\
&\quad \vee q = incomingOrderQuantity(s)
\end{aligned}
$$

$$
\begin{aligned}
&incomingSuppliesQuantity(do(a, s)) = q \equiv \\
&\quad \exists supplier\, a = deliverSupplies(supplier, q) \\
&\quad \vee q = incomingSuppliesQuantity(s)
\end{aligned}
$$

Given a GDL domain specification, successor state axioms are generated automatically by the GDL compiler. The result is a theory of the following form:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action α, characterizing *Poss*(α, s).
- Successor state axioms, one for each fluent F, stating under what conditions $F(\vec{x}, do(a, s))$ holds as function of what holds in situation s.
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

The latter foundational axioms include unique names axioms for situations, and an induction axiom. They also introduce the relation $<$ over situations. $s < s'$ holds if and only if s' is the result of some sequence of actions being performed in s, where each action in the sequence is possible in the situation in which it is performed; $s \leq s'$ stands for $s < s' \vee s = s'$. Since the foundational axioms play no special role in this paper, we omit them. For details, and for some of their metamathematical properties, see Lin and Reiter [11] and Reiter [18].

4.3 Semantics of the *ConGolog* Process Description Language

In [3], a semantics for the *ConGolog* process description language is developed within the situation calculus. This semantics, a kind of structural operational semantics [16], is based on the notion of *transitions*, i.e. "single steps" of computation. A step here is either a primitive action or testing whether a condition holds in the current situation. Two special predicates are introduced, $Final$ and $Trans$, where $Final(\sigma, s)$ is intended to say that process σ may legally terminate in situation s, and where $Trans(\sigma, s, \sigma', s')$ is intended to say that process σ in situation s may legally execute one step, ending in situation s' with process σ' remaining.

$Final$ and $Trans$ are characterized by a set of axioms, each depending on the structure of the first argument.[5] Let us only list a few of these axioms to illustrate the approach. For $Final$, we have:

- $Final(\mathbf{nil}, s) \equiv True$
  i.e., if what remains to execute is the empty process we are done;

- $Final(\alpha, nil, s) \equiv False$
  i.e., if what remains to execute is a primitive action we are not done;

- $Final([\sigma_1; \sigma_2], s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$
  i.e. a sequence can be considered done in a situation s if both components are done in s.

The axioms for $Trans$ include:

- $Trans(\alpha, s, \delta, s') \equiv Poss(\alpha, s) \wedge \delta = \mathbf{nil} \wedge s' = do(\alpha, s)$
  i.e., if we are in situation s and the process remaining is a primitive action α, we can do a transition to the situation $do(\alpha, s)$ with the empty process remaining provided that α is possible in s;

- $Trans([\sigma_1; \sigma_2], s, \delta, s') \equiv Final(\sigma_1, s) \wedge Trans(\sigma_2, s, \delta, s')$
  $\vee\, \exists \delta'. \delta = (\delta'; \sigma_2) \wedge Trans(\sigma_1, s, \delta', s')$
  i.e. a sequence $[\sigma_1; \sigma_2]$ can do a transition by performing a transition from its first component σ_1 or by performing a transition from its second component σ_2 provided that the first component is already done.

With *Final* and *Trans* in place, one can complete the semantics by defining a predicate Do, where $Do(\sigma, s, s')$ means that process specification σ, when executed starting in situation s, has s' as a legal terminating situation. The definition of Do is:

$$Do(\sigma, s, s') \stackrel{\text{def}}{=} \exists \delta. Trans^*(\sigma, s, \delta, s') \wedge Final(\delta, s')$$

[5] Note that these quantify over process specifications and so it is necessary to encode *ConGolog* process specifications as first-order terms, including introducing constants denoting variables, and so on. As shown in [5], this is laborious but quite straightforward. We omit all such details here and simply use process specifications within formulas as if they were already first-order terms.

where *Trans** is the transitive closure of *Trans*.[6] In other words, $Do(\sigma, s, s')$ holds if and only if it is possible to repeatedly single-step the process σ, obtaining a process δ and a situation s' such that δ can legally terminate in s'.

When a domain contains exogenous actions, we are usually interested in executions of the process specification where instances of the exogenous actions occur. From the GDL action declarations, one can define using a predicate Exo, which actions can occur exogenously. For our domain, we would have:

$$\begin{aligned} Exo(a) \equiv\ & \exists cust, q\ a = mkOrder(cust, q) \\ & \vee \exists supp, q\ a = deliverSupplies(supp, q) \end{aligned}$$

One can then define a special process for exogenous actions:

$$\delta_{EXO} \stackrel{\text{def}}{=} (\pi\, a.\, Exo(a)?; a)^*$$

Executing this program involves performing zero, one, or more nondeterministically chosen exogenous actions. Then we make the user-specified process δ run concurrently with δ_{EXO}:

$$\delta \parallel \delta_{EXO}$$

In this way we allow exogenous actions whose preconditions are satisfied to asynchronously occur (outside the control of δ) during the execution of δ.

A more detailed description of the *ConGolog* process language and its formal semantics appear in [3, 4]. One limitation of the semantics is that it does not handle non-terminating processes.

4.4 Using the Semantics in Verification

.

Now that we have outlined the semantics of *ConGolog* let us show how it can be used in verification. We show that our mail-order business domain specification satisfies the property that no order is shipped before payment is processed (provided that this is true initially). Formally, we want to prove that:

$$\begin{aligned} \forall s, s'.[&\forall o.\, OrderShipped(o, s_0) \supset PaymentProcessed(o, S_0)] \\ &\wedge Do(main \parallel \delta_{EXO}, S_0, s) \wedge s' \leq s \supset \\ &[\forall o.\, OrderShipped(o, s') \supset PaymentProcessed(o, s')]. \end{aligned}$$

[6] $Trans^*$ can be defined as the (second-order) situation calculus formula:

$$Trans^*(\sigma, s, \sigma', s') \stackrel{\text{def}}{=} \forall T[\ldots \supset T(\sigma, s, \sigma', s')]$$

where the ellipsis stands for:

$$\begin{aligned} &\forall s.\, T(\sigma, s, \sigma, s) \quad \wedge \\ &\forall s, \delta', s', \delta'', s''.\, T(\sigma, s, \delta', s') \wedge \mathit{Trans}(\delta', s', \delta'', s'') \ \supset\ T(\sigma, s, \delta'', s''). \end{aligned}$$

We prove this by contradiction. Suppose that there exists a situation s' that is during an execution of $main \quad || \quad \delta_{EXO}$ such that $OrderShipped(o, s')$ and $\neg PaymentProcessed(o, s')$. We can also suppose that s' is the earliest such situation, since if this is not the case, we can always move to an earlier situation. Now $s' \neq S_0$ since we are given that no order has been shipped without payment being processed initially. So $s' = do(a, s'')$ for some a and s'' and

$$OrderShipped(o, s'') \supset PaymentProcessed(o, s''),$$

since s' is the earliest situation where this doesn't hold. As well, since $\neg PaymentProcessed(o, s')$, it follows that $\neg PaymentProcessed(o, s'')$ by the successor state axiom for $PaymentProcessed$, i.e.

$$\begin{aligned}&PaymentProcessed(o, do(a, s)) \equiv \\ &\quad \exists agt\ a = processPayment(agt, o) \vee PaymentProcessed(o, s).\end{aligned}$$

Therefore $\neg OrderShipped(o, s'')$. By the successor state axiom for $OrderShipped$, i.e.

$$\begin{aligned}&OrderShipped(o, do(a, s)) \equiv \\ &\quad \exists agt\ a = shipOrder(agt, o) \vee OrderShipped(o, s)\end{aligned}$$

the only action that can cause $OrderShipped(o, s')$ to become true is $shipOrder(agt, o)$, thus $s' = do(shipOrder(agt, o), s'')$.

Now using the complete semantics of the *ConGolog* process language, it can be shown that

$$\begin{aligned}&\forall s, s'', agt, o. Do(main, S_0, s) \wedge do(shipOrder(agt, o), s'') \leq s \supset \\ &\quad PaymentProcessed(o, s'')\end{aligned}$$

i.e. the process never performs $shipOrder(agt, o)$ in a situation when payment has not been processed on order o in the situation. Intuitively, this is because the only place where $shipOrder$ appears in the process specification is in the body of the second interrupt of $runWarehouse$ and $PaymentProcessed(o)$ is one of the conjuncts of the trigger condition of the interrupt. A contradiction follows. Notice that the proof does not require anything to be known about the initial situation other than the fact that the property wasn't already false there.

5 Conclusion

The *ConGolog* framework is an attempt to develop a middle ground between state-oriented and predicate-oriented models of dynamic domains. The paper has illustrated how *ConGolog* combines elements of both approaches to support the modeling of complex dynamic domains and analyze such models through simulation and verification. Our work on applying *ConGolog* to requirements analysis and process modeling is part of a larger project dealing with process reengineering and modeling the rationale for various design alternatives [21].

The closest rival to this work is the SCR (Software Cost Reduction) framework of formal specification [8], which allows both proofs of formal properties and simulation. Unlike *ConGolog*, the SCR framework is based on a vector representation for states and a collection of finite state machines for processes. In this respect, the *ConGolog* framework is more general and more readily applicable to business process and enterprise modeling.

The most pressing task for future research is to complete the development of the *ConGolog* verification tool so that it can support a designer in verifying properties of process specifications along the lines described in sections 3 and 4. Even though *ConGolog* has been used to model and analyze several example domains, we plan to experiment with the scalability of the *ConGolog* tools by trying them out on larger and more realistic examples. We are also investigating ways of combining the *ConGolog* framework with the design rationale modeling formalism described in [21].

References

[1] D. Bjorner and C.B. Jones. *The Vienna Development Method: The Metalanguage*, volume 61 of *LNCS*. Springer-Verlag, 1978.

[2] A. Borgida, J. Mylopoulos, and R. Reiter. ...and nothing else changes: The frame problem in procedural specifications. In *Proc. ICSE-93*, 1993.

[3] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1221–1226, Nagoya, Japan, August 1997.

[4] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus: Language and implementation. In preparation, 1998.

[5] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus: Foundations. In preparation, 1998.

[6] M. Gelfond and Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(301–327), 1993.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1997.

[8] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(5), July 1996.

[9] Todd Kelley and Yves Lespérance. The Golog Domain Language: an abstract language for specifying domain dynamics. In preparation, 1998.

[10] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(59–84), 1997.

[11] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994.

[12] Fangzhen Lin and Raymond Reiter. How to progress a database. *Artificial Intelligence*, 92:131–167, 1997.

[13] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, UK, 1979.

[14] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[15] Dimitris Plexousakis. Simulation and analysis of business processes using GOLOG. In *Proceedings of the Conference on Organizational Computing Systems*, Milpitas, CA, August 1995.

[16] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Dept., Aarhus University, Denmark, 1981.

[17] Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[18] Raymond Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

[19] Steven Shapiro, Yves Lespérance, and Hector J. Levesque. Specifying communicative multi-agent systems with ConGolog. In *Working Notes of the AAAI Fall 1997 Symposium on Communicative Action in Humans and Machines*, Cambridge, MA, November 1997.

[20] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[21] Eric K.S. Yu, John Mylopoulos, and Yves Lespérance. AI models for business process reengineering. *IEEE Expert*, 11:16–23, August 1996.