Original Article

Institution of MECHANICAL ENGINEERS

Journal of RISK AND RELIABILITY

Proc IMechE Part O:
J Risk and Reliability
1–10

DOI: 10.1177/1748006X17728209
journals.sagepub.com/home/pio
SAGE

AltaRica 3.0 assertions: The whys and wherefores

Michel Batteux[1], Tatiana Prosvirnova[2] and Antoine Rauzy[3]

Abstract

In discrete event simulations, the system is assumed to change of state when and only when an event occurs. This change of state can be more or less sophisticated depending on the modeling formalism. In this article, we discuss the whys and wherefores of the fixpoint assertion mechanism introduced in AltaRica 3.0 to perform changes of states. We show how it can be used to handle complex phenomena such as change in flow directions depending on the states of components. We propose an efficient implementation of this mechanism, thanks to ideas stemmed in theoretical computer science and artificial intelligence. We compare the AltaRica 3.0 approach with alternative ones, including those of the previous versions of the language.

Keywords

Discrete event simulation, AltaRica, assertions

Date received: 11 May 2015; accepted: 21 September 2016

Introduction

In discrete event simulations, the system is assumed to change of state when and only when an event occurs. This change of state can be more or less sophisticated depending on the modeling formalism. The expressive power of the latter depends heavily on the former. A tradeoff must be found however between expressive power and algorithmic efficiency. A simulation may actually call this basic mechanism a tremendous number of times. The overall algorithmic efficiency of the simulation is thus directly related to the algorithmic efficiency of this basic brick. This may have in turn a strong influence on the quality of the results. In a Monte-Carlo simulation, for instance, within a given amount of calculation resources, the more histories can be drawn, the more accurate the results.

In this article, we discuss the whys and wherefores of the fixpoint assertion mechanism introduced in AltaRica 3.0 to perform changes of state.

AltaRica is a high-level modeling language dedicated to (probabilistic) safety analyses. Its semantics is defined in terms of transition systems: the state of the system, which is described by means of variables, is assumed to be modified when and only when an event (typically the failure of a component) occurs, that is, when a transition labeled with this event is fired. Since its very first version,[1–4] AltaRica distinguishes two types of variables: state variables and flow variables. As their name indicates, state variables are used to represent the state of the system. They can be modified only through actions (post-conditions) of transitions. Flow variables are primarily used to model information circulating between the components of a system, that is, eventually to model remote interactions between these components. The value of flow variables is calculated from the value of state variables by means of assertions. Assertions are thus applied after each transition firing.

The ability to model "simply" remote interactions is of paramount importance for safety analyses. One of the primary objectives of these analyses is actually to study the consequences of failures of individual components onto the system as a whole. Typically, one wants to model the consequences of the loss of power supply onto pumps, valves, engines, and so on. Widely used combinatorial modeling formalisms such as fault trees and reliability block diagrams rely heavily on this

[1]IRT SystemX, Palaiseau, France
[2]Computer Science Laboratory, École Polytechnique, Palaiseau, France
[3]Institutt for Produksjons- og Kvalitetsteknikk, Norges Teknisk-Naturvitenskapelige Universitet, Trondheim, Norway

Corresponding author:
Antoine Rauzy, Institutt for Produksjons- og Kvalitetsteknikk, Norges Teknisk-Naturvitenskapelige Universitet, S.P. Andersens veg 5, 7491 Trondheim, Norway.
Email: Antoine.Rauzy@ntnu.no

propagation mechanism. AltaRica assertions are very interesting with that respect because they make it possible to create complex hierarchical models from simple component descriptions just by connecting these descriptions together. Assertions can be seen as the glue between components.

The successive versions of AltaRica differ mainly by the way assertions are defined and calculated. AltaRica 3.0[5] semantics is defined in terms of guarded transition systems.[6] It introduces a fixpoint mechanism to calculate assertions. This fixpoint mechanism makes it possible to handle looped models, that is, models with flow variables that depend instantaneously (i.e. without firing of transition) on one another. Electric networks or computer networks are typical examples of systems for which it is very hard to avoid the introduction of loops. We shall explain in this article why a mechanism with at least the expressive power of fixpoints is needed in order to deal with looped systems and why fixpoint provides a minimal solution for that purpose.

Not only the fixpoint semantics for assertion provides an elegant solution to complex modeling problems but it can also be implemented in an efficient way, that is, in quasi-linear time with respect to the size of the assertion, thanks to ideas stemmed in theoretical computer science (namely, Tarjan's[7] algorithm to compute strongly connected components in graphs) and artificial intelligence (namely, unit resolution[8]). We shall show how to apply these ideas in the framework of modeling languages such as AltaRica.

The contribution of this article is as follows: first, to explain in detail and by means of examples why a fixpoint mechanism has been introduced to solve assertions in AltaRica 3.0; second, to show how this fixpoint mechanism can be implemented efficiently; third, to compare the AltaRica 3.0 approach with other approaches proposed in the literature, including those of previous versions of the language.

The remainder of this article is organized as follows. Section "Problem statement" states the problem by means of an example and reviews different state update mechanisms available in modeling languages. Section "Main ideas" discusses the main ideas behind the design and the treatment of AltaRica 3.0 assertions. Section "Assertions and their semantics" presents the AltaRica 3.0 assertions and gives their formal semantics. Section "Related works" compares the AltaRica 3.0 mechanisms with related works. Section "Conclusion" concludes the article.

Problem statement

Motivating example

Most of the industrial systems can be represented as (possibly hierarchical) networks of components. In discrete events models, states of individual components change when and only when events such as failures or repairs occur. There are in general remote interactions between components: a flow of information is circulating through the network. The term information must be taken here in a broad sense. It can be fluid, energy, data, and so on. Widely used modeling formalisms such as block diagrams and fault trees rely on the principle "local events + propagation" (see, for example, Andrews and Moss[9] and Rausand and Høyland[10] for reference textbooks).

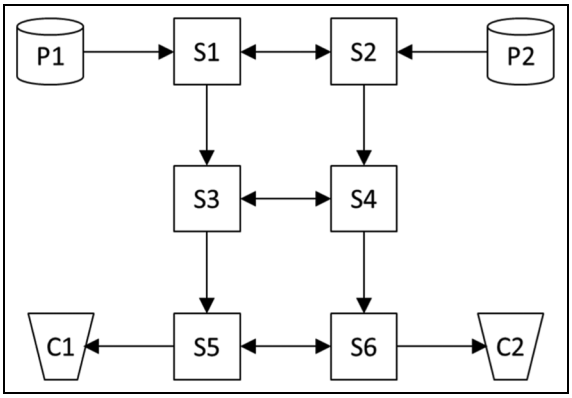

Figure 1. The network.

In fault trees and block diagrams, this propagation mechanism is quite simple. It goes one way from sources (basic events or source blocks) to targets (top events or target blocks). But there are systems for which things are more difficult. As an illustration, consider the network depicted in Figure 1 which is inspired from the fueling system of a two-engine aircraft fighter. This network is made of four types of components:

- Producers P1 and P2, represented as cylinders (in the fueling system, they are tanks);
- Consumers C1 and C2, represented as trapeziums (in the fueling system, they are engines);
- Switches S1 to S6, represented as squares (in the fueling system, they are pumps and valves);
- Directional connections between these different units, represented as arrows (in the fueling system, they are pipes).

Producers, consumers, and switches may fail and be repaired. Without a loss of generality, we can assume that connections are perfectly reliable. The problem is to assess the probability that for each consuming unit C_j, there is a working path from at least a production unit P_i to C_j.

As stated above, the problem can be casted as a "network reliability" problem for which a large literature exists.[11–13] It is however easy to complicate the formulation a bit, so that purely Boolean models are not suitable. It suffices, for instance, to introduce non-binary states to represent degradation levels or different stress of components (not to speak indeed to limited access to resources such as repair crews).

Whether there is an operating path from producers to consumers depends only on the state of components.

Depending on the operating path, flows can go in different directions (e.g. from S3 to S4 or the reverse way). Therefore, there is no simple way to propagate the availability of producers down to the consumers. For instance, it is not possible to write a fault tree and more generally a data-flow representation, that is, a set of equations in the form $x = f(y_1, \ldots, y_k)$ such that none of the y_i depends on x. Writing such set of equations would require actually to solve the problem upfront, that is, to determine paths from producers to consumers, which is precisely what the model is designed for.

A discrete event simulation of such a network needs to implement two different mechanisms: first, change of states of components under the occurrence of events (such as failures and repairs); second, propagation of information through operating components. The design of a suitable propagation mechanism is far from easy because of the loops, that is, the mutual dependencies between components.

AltaRica model

To make things a bit more concrete, we shall propose now an AltaRica 3.0 implementation of our red wire example. All components could be described by the same generic component. However, we shall assume that they are implemented by means of different components. The code for switches is given in Figure 2. It is rather simple to interpret. The state of the switch is represented by a Boolean (state) variable working. The fact that the switch is powered is represented by a Boolean (flow) variable powered. State and flow variables are declared in the same way. They are differentiated by the attribute "init" for state variables and "reset" for flow variables (the default value). A negative exponential distribution of parameter lambda is associated with the event failure. The parameter can be changed when the component is instantiated. The code declares a transition. This transition is labeled with the event failure. Its guard (precondition) is that the Boolean variable working is true. Finally, its action (post-condition) consists of setting the variable working to false.

The code for the system is given in Figure 3. After the declaration of components, the assertion is written. This assertion describes how power is propagated from producers to consumers. According to this set of equations, "X.powered" variables depend on each other (and on the "X.working" variables).

```
class Switch
   Boolean working (init = true);
   Boolean powered (reset = false);
   event failure (delay = exponential(lambda));
   parameter Real lambda = 1.0e-3;
transition
   failure: working -> working := false;
end
```

Figure 2. AltaRica 3.0 implementation of switches.

Note that the assertion would not change if components would have more complex states, typically to take into account degradation or stress levels. Note also that advanced AltaRica programmers would probably prefer to create more flow variables so not to have to use state variables outside of components. Note finally that the set of Boolean equations of Figure 3 is apparently much simpler than differential equations one would write in modeling languages such as Modelica (see, for example, Fritzson[14] for a introduction). However, in Modelica, sets of equations have to be data-flow at run time. The orientation and ordering of equations are obtained by means of a Gaussian elimination at compile time. Such a compilation technique does not work on looped networks because the direction of the flow depends on the states of components (which change with transition firings).

The question is thus how to handle the set of equations of Figure 3.

Formal statement

Let us state formally the problem as follows.

Let $S = \{s_1, \ldots, s_m\}$ be a finite set of variables. s_i are used to describe the states of individual components of a system. Each s_i takes its value into a finite or denumerable set $dom(s_i)$ called the domain of s_i. The set of the system states belongs, therefore, to the Cartesian product $dom(s_1) \times \cdots \times dom(s_n)$ of domains of the s_i. Note that it does not matter whether the state of an individual component is represented by one or more state variables.

In addition to the s_i, a second finite set $F = f_1, \ldots, f_n$ of variables is used to model the information circulating through the network of components. As for the s_i, each f_j takes its value into a finite or denumerable set $dom(f_j)$.

The value of flow variables depends on the value of state variables: we assume given an update mechanism *update* to perform the calculation, that is, $F = update(S)$.

The state of the system changes when and only when an event occurs. Thus, we have a finite set $E = \{e_1, \ldots, e_r\}$ of events, as well as a finite set

```
block System
   Producer P1, P2;
   Switch S1, S2, S3, S4, S5, S6;
   Consumer C1, C2;
assertion
   P1.powered := P1.working;
   P2.powered := P2.working;
   S1.powered := S1.working and (P1.powered or S2.powered);
   S2.powered := S2.working and (P2.powered or S1.powered);
   S3.powered := S3.working and (S1.powered or S4.powered);
   S4.powered := S4.working and (S2.powered or S3.powered);
   S5.powered := S5.working and (S3.powered or S6.powered);
   S6.powered := S6.working and (S4.powered or S5.powered);
   C1.powered := C1.working and S5.powered;
   C2.powered := C2.working and S6.powered;
end
```

Figure 3. AltaRica 3.0 implementation of the network.

$T = \{t_1, \ldots, t_p\}$ of transitions. Each transition t_i is a triple $\langle guard(S,F), e, action(S,F)\rangle$, where

- $guard(S,F)$ is a Boolean condition on the value of the variables;
- e is an event of E;
- $action(S,F)$ is a mechanism that changes the value of state variables.

For the sake of clarity, we shall denote a transition t: $\langle guard(S,F), e, action(S,F)\rangle$ as $guard(S,F) \xrightarrow{e} action(S,F)$.

The transition $t : guard(S,F) \xrightarrow{e} action(S,F)$ is fireable in a given state $\langle \bar{s}, \bar{f}\rangle$, that is, in a given assignment of the variables, if $guard(\bar{s},\bar{f}) = true$. In that case, the firing of the transition changes the state $\langle \bar{s}, \bar{f}\rangle$ of the system to the state $\langle \bar{s}', \bar{f}'\rangle$ such that $\bar{s}' = action(\bar{s},\bar{f})$ and $\bar{f}' = update(\bar{s}',\bar{f})$.

Starting from an initial state $\langle \bar{s}_0, \bar{f}_0\rangle$, it is possible to calculate the reachability graph $G = (V, E)$ of the system as follows:

- $\langle \bar{s}_0, \bar{f}_0\rangle \in V$;
- If $\langle \bar{s}, \bar{f}\rangle \in V$ and $t : guard(S,F) \xrightarrow{e} action(S,F)$ is a transition of T such that t is fireable in $\langle \bar{s}, \bar{f}\rangle$, then $\langle \bar{s}', \bar{f}'\rangle \in V$, where $\bar{s}' = action(\bar{s},\bar{f})$ and $\bar{f}' = update(\bar{s}',\bar{f})$, and $\langle \bar{s}, \bar{f}\rangle \xrightarrow{e} \langle \bar{s}', \bar{f}'\rangle \in E$.

Pedantically, G is a Kripke[15] structure. The above calculation principle is a fixpoint mechanism for it is iterated until no more vertex or edge can be added to G. The size of G can be exponentially larger than the size of the system description (but it may never be fully constructed, nor even explored).

The actions of transitions act in general very locally: they involve only a small subset of state variables (only one in the code of Figure 2). On the converse, the update function may impact many if not all flow variables. Its algorithmic efficiency is thus of primary importance.

Probability distributions can be associated with events so to give the model a probabilistic interpretation (as illustrated Figure 2). This interpretation is essentially the same as in generalized stochastic Petri nets.[16] Three types of assessment tools can then be applied: compilation into fault trees,[17] compilation into possibly partial Markov chains,[18] and of course Monte-Carlo simulation (see, for example, Zio[19] for an introductory textbook).

Main ideas

In this section, we present the main ideas behind the design and the treatment of AltaRica 3.0 assertions.

Non-determinism of value propagation

Consider again the set of equations of Figure 3. Assume that after a transition firing, all units but S4 are working correctly. Then, we can use equations to make the following deductions in order (the right-hand side number indicates which equation is applied).

P1.working	⇒	P1.powered	(1)
P2.working	⇒	P2.powered	(2)
not S4.working	⇒	not S4.powered	(6)
S1.working and P1.powered	⇒	S1.powered	(3)
S2.working and P2.powered and S1.powered	⇒	S2.powered	(4)
S3.working and S1.powered	⇒	S3.powered	(5)
S5.working and S3.powered	⇒	S5.powered	(7)
S6.working and S5.powered	⇒	S6.powered	(8)
C1.working and S5.powered	⇒	C1.powered	(9)
C2.working and S6.powered	⇒	C2.powered	(10)

Values of state variables are propagated step by step to flow variables. Each equation is used (at most) once as a deduction rule, where the right-hand side is the hypothesis and the left-hand side is the conclusion.

There is, however, a problem with the above propagation method. Consider now that after the last transition firing, all units are working correctly but S4 (as previously) and S1 (the last transition failed S1). Then, we can make the following deductions.

P1.working	⇒	P1.powered	(1)
P2.working	⇒	P2.powered	(2)
not S1.working	⇒	not S1.powered	(3)
not S4.working	⇒	not S2.powered	(6)
S2.working and P2.powered	⇒	S2.powered	(4)
S3.working and not S1.powered and not S4.powered	⇒	not S3.powered	(5)

At this point, nothing can be deduced anymore and we end up with the following simplified set of equations (by applying Boolean reduction).

7′:	S5.powered	=	S6.powered
8′:	S6.powered	=	S5.powered
9′:	C1.powered	=	S5.powered
10′:	C2.powered	=	S6.powered

This set of equations has actually two solutions: (1) all variables set to false which is the only physically realistic solution and (2) all variables set to true.

Preferred solutions and reachability in graphs

The first question is therefore whether it is possible to write a set of equations, or more generally constraints, so to keep only the physically realistic solutions in the above and other similar situations. In order word, we would like to express a preference between the two solutions. The artificial intelligence community devoted a lot of work on the representation of preferences or default values.[20] However, these works are hardly

applicable in our framework. In our case, the problem consists actually of determining whether a node is reachable from another node in a graph: a component is powered if there is at least one operating path from a source to that component.

Any graph can be seen as a binary relation $G(s, t)$ where s represents the source states and t represents the target states. Ideally, we would like to write a generic first-order formula (a predicate calculus formula) $R(s, t)$ incorporating $G(s, t)$, such that $R(s, t)$ is true if and only if the state t is reachable from the state s in the graph $G(s, t)$. However, a fundamental theorem of computational complexity theory asserts reachability is not first-order expressible.[21] In other words, theorem asserts that such a generic formula cannot exist. Predicates, logical connectives (and/or/not), and quantification over variables are not sufficient to express reachability. To express reachability, we need a form of quantification over relations as in monadic second-order logic.[22] Unfortunately, no efficient solver exists for such an expressive logic.

An idea could be to live with non-determinism for flow update, that is, to accept the two solutions, and to solve the problem using immediate events and reconfiguration transitions. Although feasible to some extent, this solution has several major drawbacks: first, it overloads the model with ad hoc events and transitions that have nothing to do with the studied system. Second, it is very inefficient. Constraint solving (that would be used to determine solutions) requires much more calculation resources than value propagation. Third, there may be many intermediate states. Consider for instance the case where components P1, P2, S3, and S4 are failed. In that case, the two subsystems S1/S2 and S5/S6/C1/C2 are isolated. Each of them has two possible states, so there are four possible states. This can be extended ad libitum. Fourth, writing immediate transitions to eliminate non-physical solutions may not be that easy. It requires actually to introduce preferences at transition level, which is moving from Charybdis to Scylla.

Choosing among the two solutions of the above system of equations requires, thus, an extra-logical preference mechanism.

Fixpoint solutions

Computational complexity theory tells us that the least mechanism at hand to solve reachability in graph is the notion of fixpoint. The idea is to iterate a calculation that accumulates reachable states until no more states can be added, that is, a fixpoint is reached. The propositional logic equipped with fixpoint mechanisms is called the μ-calculus (see, for example, Arnold and Niwiński[23] for a survey on the μ-calculus). It is extensively used in computer science, particularly in works on formal software verification and model checking.[24] Basically, the idea is to express reachability as follows

$$R(s,t) \stackrel{def}{=} G(s,t) \quad \vee \quad [\exists u, G(s,u) \wedge R(u,t)]$$

The above definition is recursive. It is solved by means of at least fixpoint mechanism: at first, the set of pairs satisfying the relation is empty, then all pairs of states (s, t) such that there is an edge from s to t are added to the set, then all pairs of states such that there exists a third state u and two edges from s to u and from u to t are added to the set, and so on, until no more pairs can be added, that is, the fixpoint is reached. Since there is finite number n of states, the longest path with no loop in the graph has length n and thus the fixpoint is reached in at most $n + 1$ steps.

An idea could be as follows: first, to reset flow variable to a default value after each transition firing; second, to use equations as value propagation rules to recalculate values of flow variables. Rules are applied until a fixpoint is reached. In our example, this default value is false for all flow variables.

AltaRica 3.0 implements actually this idea but in such a way that the value of a flow variable is calculated only once and only flow variables that need to be updated actually are, hence, reaching an optimal algorithmic complexity. This algorithmic efficiency is obtained by means of ideas stemmed from graph theory (partitioning) and artificial intelligence (unit resolution). We shall present them in the next two subsections.

Partitioning

Assume now that all units are working and that the switch S4 fails (so we are back to the situation examined above). The failure of S4 may impact units downstream, that is, S3, S5, S6, C1, and C2, but not units upstream, that is, P1, P2, S1, and S2.

The idea consists, therefore, of creating a dependency graph among the variables. The nodes of the graph are the variables. There is an edge between the node labeled with the variable x to the node labeled with the variable y if x occurs in the right-hand side member of an equation whose left member is y (there may be more than one such equation for reasons that

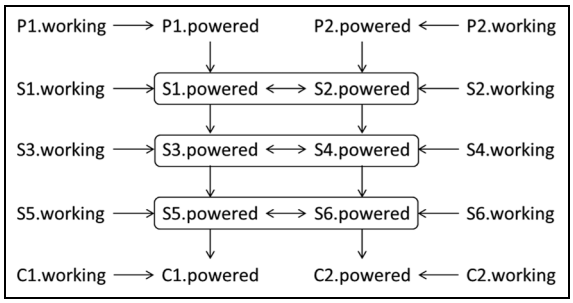

Figure 4. Dependency graph with its strongly connected components.

will be explained in the next section). The dependency graph for our example is pictured in Figure 4.

When the value of a state variable is modified by a transition firing, only the flow variables that depend (recursively) on that variable need to be updated. Moreover, the assertion can be separated into several subsets according to strongly connected components (recall that a strongly connected component C is a subset of nodes such that for any two nodes u and v in C, there is a path from s to t and a path from t to s) of the dependency graph and each of these subsets of equations can be assessed independently (in the order induced by the dependency relation).

In our example, equations defining "S3.powered" and "S4.powered" are assessed first, then those defining "S5.powered" and "S6.powered," and finally and independently those defining "C1.powered" and "C3.powered."

This partitioning technique is widely used in compiler construction, model-checking, and other computer science and engineering domains.

The calculation of strongly connected components is performed offline and is linear in the size of the dependency graph, thanks to Tarjan's[7] algorithm.

Unit resolution

The AltaRica 3.0 update mechanism works in two steps: first, values of impacted flow variables are reset to the value "unknown" and values of states variables are propagated as explained below. Second, all flow variables whose values have not been deduced are set to their default value and the equations are checked for consistency (which is possible since the value of all variables is known). If the values of the left-hand and right-hand sides of an equation differ, an error is raised (just as for division by 0 or if a variable is given a value outside of its domain).

Both steps of the update mechanism are warranted to be performed in a quasi-linear number of operations (with respect to the number of variables and equations) since once a flow variable is given a value, it keeps this value until the end.

In our example, strongly connected components are small (at most two variables), but there are indeed cases where they are much bigger. The idea consists of building a list of occurrences for each variable. When the value of the variable is modified, the equations in which this variable occurs are checked to see whether it is possible to deduce the value of another variable.

Variables whose value has been deduced (or modified) are stored into a stack. The equations to be looked at are stored into another stack. Each time the value of a variable is modified or deduced, its occurrence list is traversed and equations that are not already stacked and whose left-hand side variable is not already known are stacked. Then, each stacked equation is considered in turn.

This double-stack mechanism is an adaptation of the one used in Dowling and Gallier's[8] unit propagation (which requires only one stack). It warranties a quasi-linear complexity, although not a strictly linear one (conversely to unit propagation) for an equation can be looked at several times.

It is of primary importance to note that if the set of equations is data-flow, that is, if strongly connected components are made of only one variable at a time, the mechanism warranties the same algorithmic complexity as in data-flow languages. In other words, the increased expressive power and in particular the ability to handle looped systems is obtained without any significant overhead.

Discussion

If the AltaRica model is correctly written, the propagation mechanism cannot end up with an inconsistency. It is however easy to write an (incorrect) AltaRica model that raises a contradiction, as illustrated in Figure 5.

Initially, the state variable A is false and the flow variables B and C are free to take any value (however, if B is true, then C must also be true), so they take their default value, namely. false, and everything is all right. If the transition e is fired, then A gets true. By means of the first if-then rule of the assertion, B also gets true. Then, by means of the second if-then rule, C also gets true. But by means of the third if-then rule, C gets false. The propagation thus results in a contradiction. In this case, it would be easy to detect the problem at compile time (e.g. by trying the two possible values of A). In the general case, detecting such a potential contradiction is non-deterministic polynomial-time (NP)-hard at best, that is, computationally intractable. Therefore, we made the choice to accept incorrect models and to raise errors at run time, if any.

It is possible to write incorrect programs in any sufficiently powerful programming language. Detecting bugs at compile time is computationally intractable and in many cases even undecidable.[21] Therefore, it is accepted to live with the problem of incorrect programs and to detect issues at run time. Modeling languages do not show a different picture. A tradeoff must be

```
block IncorrectModel
   Boolean A (init = false);
   Boolean B, C (reset = false);
   event e;
   transition
      e: not A -> A := true;
   assertion
      if A then B := true;
      if B then C := true;
      if (A and B) then C := false;
end
```

Figure 5. An incorrect model.

found between detecting potential problems and spending time in looking for them.

Assertions and their semantics

In this section, we present the AltaRica 3.0 assertions together with their semantics in a structured operational semantic way.[25]

Expressions and instructions

Let C be a denumerable set of constants and V be a finite set of symbols, called variables. These variables may be of any type: Boolean, integer, real, or symbols. Let denote by *dom* a function that associates with each variable its domain (a set of values of the variable), that is, a subset of C. Expressions can be built over variables, for example, arithmetic expressions (addition, subtraction, etc.) and Boolean expressions (conjunctions, disjunctions, etc.).

A variable assignment is a function, $\sigma : V \rightarrow C$, that associates for each variable $v \in V$ its value. We say that the variable assignment σ is acceptable if $\forall v \in \mathcal{V}, \sigma(v) \in dom(v)$. Variable assignments can be extended into mapping of expressions to values, assuming a natural semantics for expressions (e.g. $\sigma(A + B) = \sigma(A) + \sigma(B)$).

Let σ be a possibly partial variable assignment, v be a variable, and finally, E be an expression. If σ does not give a value to a variable v, we write $\sigma(v) = ?$ By extension, if σ does not give a value to sufficiently many variables to evaluate the expression E, we write $\sigma(E) = ?$ The calculation of $\sigma(E)$ may raise an error, in that case, we say that σ is not acceptable for E and we write $\sigma(E) = ERROR$.

Let c be a constant, we denote by $\sigma[c/v]$ as the assignment such that

$$\sigma[c/v](w) = \begin{cases} c & \text{if } w = v \\ \sigma(w) & \text{for any other variable } w \end{cases}$$

From now, we assume that the set of variables V is a disjoint union of the set S of state variables and the set F of flow variables, $V = S \uplus F$. Moreover, we assume that each flow variable has a default value.

The set I of instructions is the smallest set such that

- If v is a flow variable and E is an expression, then "$v := E$" is an instruction;
- If C is a Boolean expression, I is an instruction, then "if C then I" is an instruction;
- If I_1 and I_2 are instructions, then so is the parallel composition "$I_1; I_2$."

An assertion is a set of instructions. It can be assimilated as the parallel composition of its instructions. However, to ensure an optimal algorithmic complexity, thanks to the mechanisms presented in the previous section, the assertion is implemented differently.

Structured operational semantics

Let A be the assertion and π be the state variable assignment obtained after the firing of a transition. Applying A consists of calculating a new variable assignment (of flow variables) τ as follows. We start by setting all state variables in τ to their values in π: $\forall v \in S \tau(v) = \pi(v)$. An assertion A extends a variable assignment τ which is total on S but possibly partial on F, according to the rules given in structural operational semantics style in Table 1.

These rules read as follows. If the conditions given above the bar are fulfilled, then the transformation given below the bar can be applied. Conditions above the bar are separated with commas, which should be thus understood as conjunctions. Transformations are made of a pattern at the left-hand side of the arrow sign ($\rightarrow$) and the transformed pattern to the right-hand side.

As an illustration, consider first rule *S*1. It reads as follows. If the variable assignment τ does not give value to the variable v (i.e. the value of v is not yet

Table 1. Semantics of assertions.

$$S1 : \frac{\tau(v) = ?, \tau(E) \neq ?, \tau(E) \in dom(v)}{\langle v := E, \tau \rangle \rightarrow \tau[\tau(E)/v]} \qquad S2 : \frac{\tau(v) = \tau(E), \tau(E) \in dom(v)}{\langle v := E, \tau \rangle \rightarrow \tau}$$

$$S3 : \frac{\tau(E) = ERROR \text{ or } \tau(E) \notin dom(v) \text{ or } \tau(v) \neq ?, \tau(E) \neq \tau(v)}{\langle v := E, \tau \rangle \rightarrow ERROR}$$

$$S4 : \frac{\tau(C) = TRUE}{\langle v := \text{if } C \text{ then } I, \tau \rangle \rightarrow \langle I, \tau \rangle} \qquad S5 : \frac{\tau(C) = FALSE}{\langle \text{if } C \text{ then } I, \tau \rangle \rightarrow \tau}$$

$$S6 : \frac{\tau(C) = ERROR}{\langle \text{if } C \text{ then } I, \tau \rangle \rightarrow ERROR}$$

$$S7 : \frac{\langle I_1, \tau \rangle \rightarrow \tau'}{\langle I_1; I_2, \tau \rangle \rightarrow \langle I_2, \tau' \rangle} \qquad S8 : \frac{\langle I_2, \tau \rangle \rightarrow \tau'}{\langle I_1; I_2, \tau \rangle \rightarrow \langle I_1, \tau' \rangle}$$

$$S9 : \frac{\langle I_1, \tau \rangle \rightarrow \langle I_1', \tau' \rangle}{\langle I_1; I_2, \tau \rangle \rightarrow \langle I_1'; I_2, \tau' \rangle} \qquad S10 : \frac{\langle I_2, \tau \rangle \rightarrow \langle I_2', \tau' \rangle}{\langle I_1; I_2, \tau \rangle \rightarrow \langle I_1; I_2', \tau' \rangle}$$

$$S11 : \frac{\langle I_1, \tau \rangle \rightarrow ERROR}{\langle I_1; I_2, \tau \rangle \rightarrow ERROR} \qquad S12 : \frac{\langle I_2, \tau \rangle \rightarrow ERROR}{\langle I_1; I_2, \tau \rangle \rightarrow ERROR}$$

determined) but gives a value to the expression E and if moreover the value of E belongs to the domain of v, then the instruction "$v := E$," transforms τ into the assignment τ' such that $\tau'(v) = \tau(E)$ and $\tau'(w) = \tau(w)$ for all variables $w \neq v$.

Now consider rule $S4$. It reads as follows. If the condition C is evaluated to $TRUE$ in the variable assignment τ, then the instruction "$v := \text{if } C \text{ then } I$," acts on τ as the instruction I.

Finally, consider rule $S7$. It reads as follows. If instruction I_1 transforms the variable assignment τ into τ', then the instruction "$I_1; I_2$," that is, the parallel composition of instructions I_1 and I_2, acts on τ as the instruction I_2 on τ'.

Let ι be an assignment that associates each variable with its initial or default value. If after applying A to π there are unevaluated variables, then all these variables are set to their default values and A is applied to τ in order to verify that all assignments are satisfied. If there is at least an assignment that is not satisfied, then an error occurs.

The update mechanism of section "Formal statement" is the function $update(A, \iota, \tau)$ that

- Extends the partial variable assignment τ by the instruction A according to the rules given in Table 1;
- Completes τ by setting all unassigned flow variables to their default values $\forall v \in F : \tau(v) = ?/\tau(v) = \iota(v)$ and checks, still by means of rules of Table 1, that there is no contradiction.

Discussion

Rules of Table 1 are thus used into two different contexts. First, when the value of not all of the variables is determined, in order to determine more values. Second, when the value of all variables is determined, to check there is no contradiction.

It is easy to show by structural induction that if the assignment is complete (second use), at least one rule is applied to each possible instruction, that is, the checking process will reach a conclusion is any case. It is also easy to show, still by structural induction, that all possible transformations are performed in the first use. There may be, however, some non-determinism in both cases due to parallel composition. The semantics does not say which instruction is evaluated first. Of course, tools make a choice. But the analyst cannot rely on the particular choice made by a particular model to design her or his models. To put it in another way, to be correct, a model must give the same result whether the first instruction is evaluated first or the second one.

We come back here to section "Discussion." It would be great to be able to check potential non-determinism at the compile time. Unfortunately, as AltaRica 3.0 is a very expressive language, such a check would be extremely resource consuming, if not undecidable.

Note, however, that it would be possible to modify the rules so as to take into account the particular policy of a particular tool, for example, to evaluate the first instruction first. This would remove the non-determinism. The semantics of AltaRica 3.0 does not make this choice. One of the reasons is that it would rely on the order of declarations. But in the context of model-based safety assessment, where graphical interfaces are used to author models, the analyst may not have the control on this order. Therefore, it is much better not to rely on any particular tool policy to design models.

Related works

The description of systems given section "Formal statement" is very general since it does not say how the actions and update functions are actually described and calculated. Many modeling formalisms can actually be seen as particular cases of this description.

It is obvious for combinatorial formalisms such as fault trees and block diagrams. In fault trees, basic events can be seen as two state components with a Boolean output flow set to true when the component is failed and false otherwise. Gates are just propagating values bottom-up. Similarly, in block diagrams, basic blocks can be seen as two state components with an input flow and an output flow. The output flow is true if and only if the input flow is true and the component is working. Hierarchical blocks propagate values from sources to sinks of the network. Extensions of block diagrams with multi-valued flows such as in HiP-HOPS[26] work the same way.

It has also been recently showed that dynamic fault trees (as introduced, for instance, in Dugan et al.[27]) and Boolean driven Markov processes[28] can be encoded within this formalism as well.[29]

Reliability networks[11,12] make it possible to handle looped systems under the condition that components are binary (working or failed) and flow variables are Boolean. Dedicated algorithms have been proposed to handle this kind of models[13,30] but they do not scale well.

If we look now at modeling formalisms directly relying on state machines, we can see that flow propagation is extensively used in data-flow modeling languages such as Lustre,[31] SAML,[32] and AltaRica data-flow,[3,4] the former version of the AltaRica language. These languages make it possible to assemble components in an effortless way, just like a Lego construction. They can be seen as powerful extensions of block diagrams. The data-flow requirement imposes that there exists a total order over the variables such that if $x < y$, then the value of x does not depend on the value of y. It is therefore possible to update the values of variables according to this order (i.e. from sources to sinks). This mechanism is algorithmically efficient, but cannot be used to handle looped systems with systems of equations such as the one in Figure 3.

Two main alternative patterns have been proposed in the literature to model remote interactions between components:

- Shared variables, such as in Lynch and Tuttle's[33] input/output automata;
- Synchronization of events such as in Arnold–Nivat automata.[34]

Other formalisms such as Harel's[35] state charts or some variants of generalized stochastic Petri nets[16,36] mix these two approaches in different ways. With the above mechanisms, propagations of values have to be programmed "by hand," that is, by writing explicitly all instructions (or synchronizations) that perform these propagations. These mechanisms are thus not sufficient to handle remote interactions in a simple way because they are not declarative. Moreover, as we shall see in the next section, they cannot be used to handle looped systems (but of course by calculating beforehand working paths).

In the first version of AltaRica,[1,2] as it is still implemented in tools developed by LaBRI's team,[37] the assertion was seen as a constraint and flow variables are updated by constraint resolution. Constraint solving is probably too costly to be used at industrial scale. It would be, for instance, very inefficient to call a constraint solver at each step of a stochastic simulation. Moreover, as we explained in section "Non-determinism of value propagation," constraint solving is not suitable to handle looped systems. To conclude this brief overview, we should mention two related modeling formalisms.

First, flow propagation and looped systems can be handled to some extent in the Figaro modeling language.[38] Figaro technology is close to those of expert systems and de Kleer's assumption truth maintenance systems.[39] Bouissou and colleagues[40,41] sketch the syntax and semantics of the language. Some techniques are common to both languages (Figaro and AltaRica): the distinction between two types of variables, the use of dependency graphs, and the definition of the semantics of models in terms of reachability graph. The two last points are actually common to many programming and modeling languages. It remains that AltaRica syntax and semantics make, in authors' opinion, a much clearer distinction between transitions and flow updates with fewer and more elegant constructs. Moreover, and that is what this article about, assertions are handled in a very efficient way in AltaRica 3.0, while keeping a high expressive power.

Second, the idea to separate the description of states of components (and transitions between these states) from the description of (stateless) interactions (the glue) between components is at the core of BIP algebra.[42] We are deeply convinced that Sifakis et al.'s work is at least a great source of inspiration for modeling languages dedicated to safety and reliability engineering.

Conclusion

In this article, we presented AltaRica 3.0 assertions. We explained why the fixpoint mechanism introduced in AltaRica 3.0 provides the language the unique feature of being able to handle looped systems. We showed that this expressive power comes with an excellent algorithmic efficiency, thanks to compilation techniques and ideas stemmed in artificial intelligence and automated deduction. More exactly, the increased expressive power and in particular the ability to handle looped systems is obtained without any significant overhead for data-flow models.

The ideas presented here are worth not only for AltaRica but also for any modeling language dedicated to discrete event simulation.

The authors are convinced that modeling languages dedicated to discrete event simulation and probabilistic safety analyses should be studied in a systematic way. Mechanisms that are presented here should be seen as basic bricks for these languages.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

References

1. Point G and Rauzy A. AltaRica constraint automata as a description language. *J Eur Syst Autom* 1999; 33(8–9): 1033–1052.
2. Arnold A, Griffault A, Point G, et al. The AltaRica formalism for describing concurrent systems. *Fund Inform* 2000; 40: 109–124.
3. Rauzy A. Mode automata and their compilation into fault trees. *Reliab Eng Syst Safe* 2002; 78(1): 1–12.
4. Boiteau M, Dutuit Y, Rauzy A, et al. The AltaRica data-flow language in use: modeling of production availability of a multi-state system. *Reliab Eng Syst Safe* 2006; 91(7): 747–755.
5. Prosvirnova T, Batteux M, Brameret PA, et al. The AltaRica 3.0 project for model-based safety assessment. In: *Proceedings of the 4th IFAC workshop on dependable control of discrete systems (DCDS'2013)*, University of York, York, 4–6 September 2013, pp.127–132. Luxemburg: International Federation of Automatic Control.
6. Rauzy A. Guarded transition systems: a new states/events formalism for reliability studies. *Proc IMechE, Part O: J Risk and Reliability* 2008; 222(4): 495–505.
7. Tarjan RE. *Data structures and network algorithms*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.
8. Dowling WF and Gallier JH. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J Logic Program* 1984; 1(3): 267–284.

9. Andrews JD and Moss RT. *Reliability and risk assessment*. Hoboken, NJ: John Wiley & Sons, 1993.
10. Rausand M and Høyland A. *System reliability theory: models, statistical methods, and applications*. 2nd ed. Hoboken, NJ: Wiley-Blackwell, 2004.
11. Colbourn CJ. *The combinatorics of network reliability*. New York: Oxford University Press, 1987.
12. Shier DR. *Network reliability and algebraic structures*. Oxford: Oxford Science Publications, 1991.
13. Madre JC, Coudert O, Fraïssé H, et al. Application of a new logically complete ATMS to digraph and network-connectivity analysis. In: *Proceedings of the annual reliability and maintainability symposium (ARMS'94)*, Anaheim, CA, 24–27 January 1994, pp.118–123. New York: IEEE.
14. Fritzson P. *Principles of object-oriented modeling and simulation with Modelica 2.1*. Hoboken, NJ: Wiley-IEEE Press, 2003.
15. Kripke S. Semantical considerations on modal logic. *Acta Philos Fenn* 1963; 16: 83–94.
16. Ajmone-Marsan M, Balbo G, Conte G, et al. *Modelling with generalized stochastic Petri nets* (Wiley series in parallel computing). New York: John Wiley & Sons, 1994.
17. Prosvirnova T and Rauzy A. Automated generation of minimal cut sets from AltaRica 3.0 models. *Int J Crit Comput Base Syst* 2015; 6(1): 50–79.
18. Brameret PA, Rauzy A and Roussel JM. Automated generation of partial Markov chain from high level descriptions. *Reliab Eng Syst Safe* 2015; 139: 179–187.
19. Zio E. *The Monte Carlo simulation method for system reliability and risk analysis* (Springer series in reliability engineering). London: Springer, 2013.
20. Besnard P. *An introduction to default logic* (Symbolic computation). Berlin: Springer, 1989.
21. Papadimitriou CH. *Computational complexity*. Boston, MA: Addison-Wesley, 1994.
22. Courcelle B. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inform Comput* 1990; 85(1): 12–75.
23. Arnold A and Niwiński D. *Rudiments of calculus* (Studies in logic and the foundations of mathematics). Amsterdam: North Holland, 2001.
24. Clarke EM, Grumberg O and Peled DA. *Model checking*. Cambridge, MA: MIT Press, 2000.
25. Plotkin GD. The origins of structural operational semantics. *J Logic Algebr Progr* 2004; 60–61: 3–15.
26. Adachi M, Papadopoulos Y, Sharvia S, et al. An approach to optimization of fault tolerant architectures using HiP-HOPS. *Software Pract Exper* 2011; 41: 1303–1327.
27. Dugan JB, Bavuso SJ and Boyd MA. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE T Reliab* 1992; 41(3): 363–377.
28. Bouissou M and Bon JL. A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes. *Reliab Eng Syst Safe* 2003; 82(2): 149–163.
29. Rauzy A and Blériot-Fabre C. Towards a sound semantics for dynamic fault trees. *Reliab Eng Syst Safe* 2015; 142: 184–191.
30. Rauzy A. A new methodology to handle Boolean models with loops. *IEEE T Reliab* 2003; 52(1): 96–105.
31. Halbwachs N, Caspi P, Raymond P, et al. The synchronous data flow programming language LUSTRE. *P IEEE* 1991; 79(9): 1305–1320.
32. Güdemann M and Ortmeier F. A framework for qualitative and quantitative formal model-based safety analysis. In: *Proceedings of the IEEE 12th high assurance systems engineering symposium (HASE 2010)*, San Jose, CA, 3–4 November 2010, pp.132–141. New York: IEEE.
33. Lynch NA and Tuttle MR. An introduction to input/output automata. *CWI Quart* 1989; 2: 219–246.
34. Arnold A. *Finite transition systems* (series ed CAR Hoare). Upper Saddle River, NJ: Prentice Hall, 1994.
35. Harel D. Statecharts: a visual formalism for complex systems. *Sci Comput Program* 1987; 8(3): 231–274.
36. Dutuit Y, Châtelet E, Signoret JP, et al. Dependability modelling and evaluation by using stochastic Petri nets: application to two test cases. *Reliab Eng Syst Safe* 1997; 55(2): 117–124.
37. Griffault A and Vincent A. The Mec 5 model-checker. In: *Proceedings of the 16th international conference on computer aided verification (CAV 2004) (vol. 3114 of lectures notes in computer science)*, Boston, MA, 13–17 July 2004, pp.488–491. Berlin; Heidelberg: Springer-Verlag.
38. Bouissou M, Bouhadana H, Bannelier M, et al. Knowledge modelling and reliability processing: presentation of the Figaro language and associated tools. In: *Proceedings of the IFAC international conference on safety of computer control systems (SAFECOMP'91)* (ed JF Lindeberg), Trondheim, 30 October–1 November 1991, pp.69–75. Oxford: Pergamon Press.
39. De Kleer J. An assumption-based TMS. *Artif Intell* 1986; 28(2): 127–162.
40. Bouissou M, Humbert S, Muffat S, et al. KB3 tool: feedback on knowledge bases. In: *Proceedings of the European safety and reliability conference (ESREL'2002)*, Lyon, 18–21 March 2002, pp.114–119. Bagneux Cedex: Institut de Sûreté de Fonctionnement (ISdF).
41. Bouissou M and Houdebine JC. Inconsistency detection in KB3 models. In: *Proceedings of the European safety and reliability conference (ESREL'2002)*, Lyon, 18–21 March 2002, pp.754–759. Bagneux Cedex: Institut de Sûreté de Fonctionnement (ISdF).
42. Bliudze S and Sifakis J. The algebra of connectors—structuring interaction in BIP. *IEEE T Comput* 2008; 57(10): 1315–1330.