An agent model for fault-tolerant systems

Avelino Francisco Zorzo
Faculty of Informatics – PUCRS
6681, Ipiranga Avenue
90619-900 Porto Alegre, Brazil
zorzo@inf.pucrs.br

Felipe Rech Meneguzzi
HP/PUCRS
6681, Ipiranga Avenue
90619-900 Porto Alegre, Brazil
fmeneguzzi@terra.com.br

ABSTRACT

This paper describes the use of fault tolerance in a multi-agent system. Such an approach is based on the modeling of autonomous agents with planning capabilities. These capabilities are used by the agent to recover from faults occurring in its surrounding environment, e.g. hardware faults, or in its internal representation thereof, e.g. software faults. The expected fault-tolerant behavior is tested using fault injection either in the system described by the agent or in the environment in which the agent (system) is embedded into.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent agents*; D.4.5 [**Software**]: Operating Systems—*Fault-tolerance*

General Terms

Reliability, Algorithms, Languages

Keywords

CA actions, BDI Model, Planning, Declarative Goals

1. INTRODUCTION

Various techniques have been proposed in order to provide dependability for distributed systems developed using what now are standard programming techniques like Object Orientation (OO) [23]. Meanwhile, Agent Oriented Programming (AOP) [13] is being proposed as a new approach to the development of distributed systems. Its main appeal resides in it being a powerful tool that models system behavior in terms of interactions among autonomous entities.

As AOP progresses and new methodologies of agent-based software are created [7], several authors have already expressed concerns on the use of fault tolerance in Multi-Agent Systems (MAS) [11, 14]. While MAS are more flexible than traditional distributed systems due to its ability to reason about unpredicted situations and more reliable due to the fact that agents are easily replicated, failures in the system itself and whatever it controls are still possible. Therefore, it is important for a MAS to have some kind of mechanism that allows a group of agents performing some task to try to bring the system to a stable state in case of a failure, or otherwise notify the system about catastrophic failures.

Considering techniques of fault-tolerant software design [3, 17], agents can be easily related to them in that they provide a transparent interface for the implementation of multiple variations in the strategies for achieving the same result, either concurrently, using cooperating agents, or sequentially, using agents that assume the role of a previously faulty agent. Agent-based systems are also closely related to systems modeled through Coordinated Atomic (CA) actions [22], as they define a model of problem solving based on the interaction of cooperating processes.

In this paper we propose a way to implement fault-tolerant systems using an agent-based approach [15]. In particular the proposed agent model is described in terms of *declarative* goals [20]. Using a declarative goal semantics to drive its reasoning process allows an agent to decouple plan formation and execution from goal achievement [5], which enables the agent to build plans at runtime [15]. This type of description is less prone to specification errors since the designer is specifying **what** must be accomplished rather than **how** to do it. Furthermore, the goal to be accomplished by the system is also the standard against which fault-tolerant mechanisms will check the results generated by the agent, making post-condition verification more straightforward, whereas the agent planning capability provides a powerful resource that can be taken advantage of in the event faults are detected by the agent [5].

Fault-tolerant systems modeled as autonomous agents also allow the simulation of faults in terms of interactions with the surrounding environment, which in turn allows the observation of system behavior under these conditions. We, therefore, use a fault-injection technique to show how an agent can tolerate faults introduced in its surrounding environment as well as in its internal state.

2. RELATED WORK

2.1 BDI Agents

As computer systems became more complex, abstraction mechanisms for these systems were developed. One such mechanism that is becoming increasingly accepted is the notion of Computer Agents [8], so far as to be proposed as an alternative to the Turing Machine as an abstraction for the notion of computation [19].

In the context of multi-agent systems research, one of most widely known and studied models of deliberative agents uses Beliefs, Desires and Intentions (BDI) as abstractions for the description of a system's behaviour. This model was originated by a philosophical model of human practical reasoning [4], later formalized [6] and improved towards a more complete computational theory [9, 21].

The selection of a course of action an agent will take in order to satisfy its objectives, *i.e.* given an environment and a set of objectives, determine whether the agent is capable of satisfying its objectives through some sequence of actions, is one of the most important processes of the BDI model.

A number of BDI agent implementations, in particular those based on PRS [9], adopt a procedural semantics to agent goals in order to drive agent reasoning, rather than *declarative goals* [20]. The usage of procedural goals in these systems was motivated by a desire to achieve practical runtime performance in these systems, as reasoning about procedural goal is quicker than for declarative goals.

Declarative goals are, nevertheless, an important component of intelligent agent systems as it allows an agent to reason about the goals themselves [20]. An agent that has information regarding *what* it is trying to accomplish is able to determine which of its goals can be achieved, which ones are impossible, and infer the relationship among them [20]. Moreover, declarative goals allow for the decoupling of plan execution and goal achievement [5], which in turn enable an agent to search for alternate plans in order to fulfill a goal that could not be achieved due to the failure of an attempted plan [15, 18].

2.2 Fault-tolerant mechanisms

System programmers usually write programs under the optimistic assumption that, after realizing a set of tests, nothing will go wrong when a system is deployed. When something that had not been foreseen happens the system will fail in an unexpected way. This is not acceptable in many situations, e.g. in safety-critical systems. To improve reliability, it is important that this type of situation is treated appropriately.

During the past years several mechanisms were proposed in order to provide fault-tolerant behavior to software components, e.g. Recovery Block [17], NV programming [3], CA Action [22]. Although these mechanisms have been used in the past years, approaches to determine whether their behavior is suitable for dealing with environment or system faults is not clear. This may cause problems when designing a system that uses these mechanisms. Therefore it is important to define a new way of testing such mechanisms.

2.3 Fault Injection

Fault Injection (FI) is a testing technique based on the deliberate introduction and/or simulation of faults into a system and the observation thereof in order to verify system behavior under these circumstances [12]. FI testing involves subjecting the target system to faults supported by its fault-model and observe how the system reacts under these conditions. Such process is considered an important tool within the development of fault-tolerant systems as its resulting data can be used in the determination of various dependability measures, such as error detection coverage and the efficiency of a given set of fault-tolerant mechanisms [2].

This strategy can be applied at different phases within the development of a given system. In *Simulation-Based* FI, a system is evaluated at its conceptual and design stages prior to the actual implementation, and thus requires an accurate specification of the target system and its failure modes. Once a prototype is built it can be tested using *Prototype-Based* FI. This is accomplished either at software level and hardware level or both and consists of subjecting the prototype to emulated faults in order to observe its behavior. After a concrete system is deployed, actual operational data can be analyzed using *Measurement-Based* FI, which provides statistical data regarding field-observed error conditions [12].

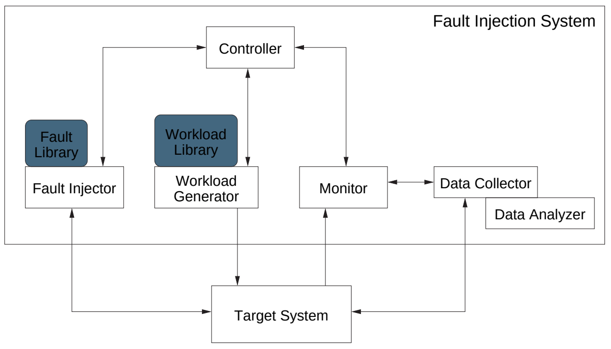


Figure 1: Typical components of an FI environment.

A possible architecture for FI environment is composed of the Target System and a Fault Injector, Fault Library, Workload Generator, Workload Library, Controller, Monitor, Data Collector and Data Analyzer (Figure 1) [12]. In such an environment, the Fault Injector injects faults into the Target System while it is processing input from the Workload Generator. The Target System's behavior is tracked by the Monitor while relevant data is gathered by the Data Collector and later examined by the Data Analyzer. The interaction among these components is coordinated by the Controller. Such an environment can be used as a reference point in the construction of new FI tools (Section 4).

3. THE X²BDI AGENT MODEL

Extended Executable BDI or **X^{2}BDI** is an extension of X-BDI [15], which is an ELP-based agent model that uses an external planning function to perform means-ends reasoning. The cognitive structure of **X^{2}BDI** has the traditional components of BDI agents, *i.e.* a set of Beliefs, Desires and Intentions. It also has a set of time axioms inherited from X-BDI. An **X^{2}BDI** agent is composed of the same components as its predecessor plus a propositional planning function conforming to the formalism described in [15]. An agent description contains the following elements: *i)* A set of actions that essentially specify the abilities possessed by an agent. An action is comprised of a set of pre-conditions that states when it is possible for the agent to execute the action and a set of effects that describe the result of the execution of that action; *ii)* A set of desires that specify a set of possible goals the agent might try to accomplish. A desire is comprised of a pre-condition that states when that goal becomes relevant to the agent. It also has a priority value used by the agent to resolve which goals to pursue when multiple goals become relevant but are not consistent; *iii)* A set of initial beliefs used to initialize the knowledge of the agent in a particular domain.

The set of beliefs is a formalization of facts in ELP, whose consistency is maintained by means of a program revision process performed in ELP by the SLX procedure [1]. From the agent's point of view, it is assumed that its beliefs are always consistent. Every desire in an $\mathbf{X^2BDI}$ agent is conditioned by a conjunction of literals called *Body*, which specifies the pre-conditions that must be satisfied in order for an agent to desire a property. Desires may be specified to be valid only in a specific moment, or whenever its pre-conditions are valid. Desires also have a priority value used in the formation of an order relation among desire sets. There are two possible types of intentions: Primary Intentions, which refer to the intended properties, and Relative Intentions, which refer to actions able to bring about these properties. An agent may not intend something in the past, that is already true, or is impossible, *i.e.* there must be at least one plan available to the agent whose result is a world state where the intended property is true.

The process of modeling $\mathbf{X^2BDI}$ agents to solve problems is very similar to that of modeling STRIPS problems [10]. The first step is the definition of the problem elements in terms of first order literals. Properties about objects in a world are represented as logic *predicates* applied to *terms* representing the objects themselves. For example if we want to say that an object `a` is a metal plate, we write `metalPlate (a)`, and, if want to say that `a` is rusted we write `rusted(a)` . Besides objects and their properties, an agent also must know the set of *operators* with which it can interact with the world. Operators are tuples $\langle pre, post \rangle$ where *pre* denotes a conjunction of pre-conditions that must be true prior to the execution of the operator and *post* denotes a conjunction of literals that will become true once the operator has been executed. Once a definition for the problem objects, properties and world manipulation operators has been reached, the agent itself is modeled. The information regarding the world defined in the first step is stored in the agent's *beliefs*, while the agent purpose in the given world is modeled through its *desires*, that represent world states that the agent will try to achieve using the knowledge contained in its beliefs.

The $\mathbf{X^2BDI}$ reasoning process initiates with the selection of Eligible Desires, which represent the unsatisfied desires whose pre-conditions have been satisfied. The elements of this set are not necessarily consistent among themselves. Candidate Desires are then generated, which represent a set of Eligible Desires that are both consistent and possible and will be later adopted as Primary Intentions. In order to satisfy the properties represented by Primary Intentions, the planning process generates a sequence of temporally ordered actions that constitute the Relative Intentions.

The process of selecting Candidate Desires seeks to choose a subset of Eligible Desires that contains only those that are internally consistent and possible, *i.e.* desires of properties P that can be simultaneously satisfied through a sequence of actions. $\mathbf{X^2BDI}$ uses an external planning function, thus separating the planning process previously hard-coded within X-BDI. A set of Candidate Desires is the subset of Eligible Desires with the greater priority value, and whose properties can be satisfied. Satisfiability is verified through the execution of a propositional planner that processes a

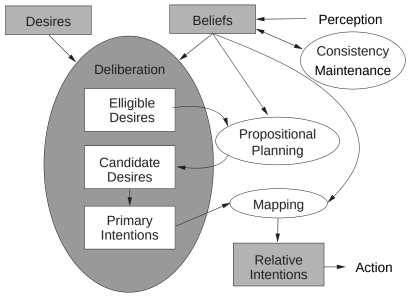


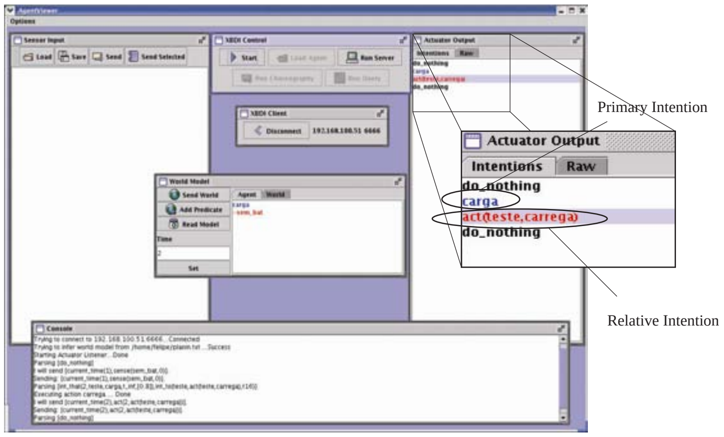
Figure 2: $\mathbf{X^2BDI}$ overview.

planning problem where the initial state contains the properties that the agent believes at the time of planning. The P properties present in the Candidate Desires are used to generate the set Primary Intentions. Primary Intentions represent the agent's commitment to achieving a set of objectives for which a course of action has been found. Relative Intentions correspond to the temporally ordered steps of the concrete plans generated to satisfy the agent's Primary Intentions. The notion of agent commitment results from the fact that Relative Intentions must be non-contradictory regarding Primary Intentions. The computational effort and the time required to reconsider the whole set of intentions of a resource-bounded agent is generally significant regarding the environment change ratio. Therefore, intention reconsideration should not occur constantly, but only when the world changes in such a way as to threaten the plans an agent is executing or when an opportunity to satisfy more important goals is detected. As a consequence, $\mathbf{X^2BDI}$ uses a set of reconsideration "triggers" generated when intentions are selected, and causes the agent to reconsider its course of action when activated [15].

4. FI USING *AGENTVIEWER*

A tool was initially created to ease the process of modeling and testing $\mathbf{X^2BDI}$ agents through a graphical tool that allows its start up and configuration as well as the remote interaction with a running agent [15]. This tool is called *AgentViewer* (Figure 3). Its main features are: control of the agent kernel execution, communication with an agent via sockets and the representation of the world model which an agent is interacting with.

It is important to point out that, although it may be argued that modifying sensor data being supplied to the agent would not represent fault injection *per se*, we advocate this modality of input manipulation as being an actual instance of fault-injection. The main argument against our brand of fault-injection is that the agent input data being modified is part of the normal system execution, and as such, it would not actually represent abnormalities within its lower level components (at machine instruction level). On the other hand, the arbitrary manipulation of the predicates used by the agent in its reasoning process could be the result of some kind of lower level bit flip, or the manifestation of a malfunction within sensor hardware. Thus we believe that, as *AgentViewer* is a simulation tool rather than being necessarily the deployment hardware platform, modifying the predicates over which the agent performs its reasoning process is a valid form of fault-injection.



```
Trying to connect to 192.168.100.51 6666 ... Connected
Trying to infer world model from /home/felipe/plann.txt ... Success
Starting Actuator Listener ... Done
Parsing [do_nothing]
I will send [current_time(1),sense(sem_bat,0)]
Sending: [current_time(1),sense(sem_bat,0)]
Parsing [int_that(2,teste,carga,t_inf,[0.8]),int_to(teste,act(teste,carrega),r16)]
Executing action carrega ... Done
I will send [current_time(2),act(2,act(teste,carrega))]
Sending: [current_time(2),act(2,act(teste,carrega))]
Parsing [do_nothing]
```

Figure 3: The *AgentViewer* tool.

In light of the classification presented in Section 2.3, our tool could be employed in all development phases of a given agent, thus allowing Simulation-Based, Prototype-Based as well as Measurement-Based fault injection. Considering the declarative nature of X-BDI agents, Simulation-Based and Prototype-Based fault injection would not be easily distinguishable as the X-BDI interpreter directly executes agent specifications. The main difference between an emulated agent run and its operation in a deployed system lies in what is connected to the the agent's sensors.

4.1 World Model

Besides easing the interaction with an **X^{2}BDI** agent, *AgentViewer* is capable of manipulating world model descriptions, which the user can freely manipulate and send to the agent to observe its behavior. *AgentViewer* can maintain two such descriptions: the *world model*, and the *agent world view*. The *agent world view* represents the world model as seen by the agent, and the *world model* represents the "real" world. Modifications performed over the *world model* represent changes in the actual environment with which the agent is interacting, whereas modifications performed over the *agent world view* only affects the information that is available to the agent, which can have no direct connection to the reality whatsoever, thus feeding the agent with faulty information. It is also possible for the user to interfere with the actions performed by the agent over the real world, causing arbitrary failures with regards to the results expected by the agent. This functionality is controlled through the `World Model` window (Figure 3), which represents valid world properties in different colors.

The *agent world view* maintained by *AgentViewer* is inferred at the moment in which it connects to an **X^{2}BDI** agent. At this time, the tool loads the last planning problem created by the agent in its communication with the planning module. In that problem, the start state represents all the properties believed by the agent in that deliberation cycle. In case the tool cannot infer a world model through this file, the user is responsible for defining a world model.

Once a representation of the *agent world view* is loaded, the user can manipulate it, and send it to the remote agent. These functionalities can be accessed through the following buttons:

- `Send World`: Sends the world model to the agent;
- `Add Predicate`: Adds user specified predicate to the selected world model, this predicate can either be positive or negative (indicated by the `-` symbol, or the reserved word `not`). All unspecified predicates are considered false by default;
- `Read Model`: Replaces the current world model for a new one specified in a file containing a user-specified STRIPS problem;
- `Time/Set`: Specifies an integer value denoting the current moment in time in the world model. Every time the world model is sent to the agent this value is incremented;
- Besides these buttons, the user can use Context Menus to delete or negate a selected predicate. When the program receives the result of an agent's deliberation through the `Actuator Output` window, the user can use Context Menus in that window to either execute the actions selected by the agent over *AgentViewer*'s world model or cause them to fail, the agent can be notified of the success or failure of its actions or not.

Through this functionality *AgentViewer* not only provides greater flexibility on agent testing, but it also provides a fault-injection environment for agent testing. In comparison to the environment described in Section 2.3, *AgentViewer* does not implement all of the functionalities contained in the reference architecture of Figure 1; the Sensor Input window is analogous to the Workload Generator and Library, while Fault Injector functionality is provided by the World Model window and partially by the Actuator Output window as it allows Agent Actions to be overridden and forced to fail, such functionality still lacks a Fault Library; Monitoring and Data Collection functions are provided by the Actuator Output window.

4.2 Injecting faults

Through the `World` tab in the `World Model` window, the user can arbitrarily change the state of any property of the agent's environment. Therefore, he can cause several types of faults within the world upon which an agent is operating. Once the agent is aware of these faults, it will try to cope with it in its deliberative process and bypass some or all of the resulting errors caused by it. By modifying the world model the user is injecting faults into the system for the agent to deal with.

The user can also modify the information that is sent to the agent's sensors while not modifying the actual world through the `Agent` tab. By feeding faulty information about the world state, the user causes the agent to make decisions based on false assumptions, thus injecting faults into the agent itself. Using this approach, one can test the agent's ability to degrade gracefully or cope with the resulting errors, while trying not to cause a system failure.

The possibility of injecting faults in the agent provided by the *AgentViewer* tool makes its fault injection different from [22]. In that work faults were injected through a fault interface into the environment with which the control program interacted, therefore only environment faults were possible whereas faults in the control software could not be injected. Furthermore, only the faults defined in the interface could be

Predicate	Meaning (denotes that)
failed (X)	component X failed
empty(X)	component X is empty
plate(P)	P is a sheet metal plate
robot(R)	R is a Robot within the cell
arm(A,R)	A is an arm of Robot R
table(T)	T is a feeding table
press(P)	P is a metal plate press
depositBelt(D)	D is a deposit belt
loaded(X,P)	component X is loaded with plate P
done(P)	a plate P has been processed

Table 1: Predicates and corresponding meaning.

injected. In our approach, which can be used to design and verify any similar application, the system verifier can inject faults either in the environment, by adding new predicates in the *world model*, or in the control system, by adding new predicates to the *agent world view*.

5. A PRODUCTION CELL CASE-STUDY

In order to check the ability of an **X^{2}BDI** agent to function in the event of faults, we have modeled a production cell in which faults in its components are possible. In [22] a design for a production cell is composed of a Deposit Belt, two Presses for the processing of sheet metal plates, a Robot with two perpendicular arms intended to move components within the cell and a table where components are placed (Figure 4). Plates entering the cell are placed in the table and moved by one of the robot's arms to a press where it is processed. Once a plate has been processed, it is moved by the robot into the deposit belt, where it is moved off the cell. Any component within the cell can fail at any time. These components are modeled with the following predicates:

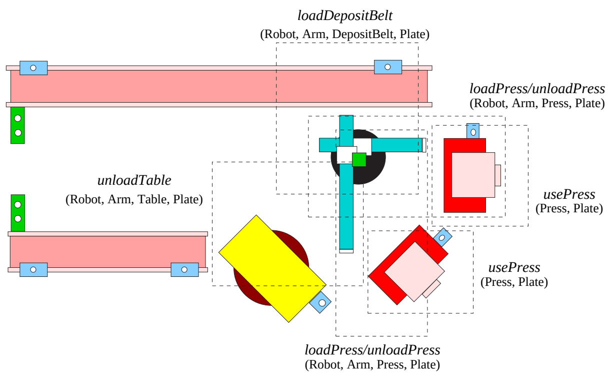


Figure 4: A fault-tolerant Production Cell.

The interaction among the components are modeled as STRIPS operators that are analogous to CA actions (for a more thorough description of CA actions, see [22]). The operators defined in this domain are the following:

- unloadTable(R, A, T, P), with pre-conditions plate(P), robot(R), arm(A,R), empty(A), table(T), loaded(T, P), −failed(A), −failed(R) and −failed(T), and effects empty(T), −loaded(T, P), −empty(A) and loaded(A, P), represents the unloading of plate P from table T by the arm A of robot R;
- loadPress(R, A, Pr, P), with pre-conditions plate(P), robot(R), arm(A, R), loaded(A, P), press(Pr), empty(Pr), −failed(A), −failed(R) and −failed(Pr), and effects −loaded(A, P), empty(A), loaded(Pr, P) and −empty(Pr), represents the loading of plate P in press Pr by arm A of robot R;
- unloadPress(R, A, Pr, P), with pre-conditions plate(P), robot(R), arm(A, R), empty(A), press(Pr), loaded(Pr, P), −failed(A), −failed(R) and −failed(Pr), and effects loaded(A, P), −empty(A), −loaded(Pr, P) and empty(P), represents the unloading of plate P from press Pr by arm A of robot R;
- usePress(Pr, P), with pre-conditions plate(P), press(Pr), loaded(Pr, P) and −failed(Pr), and effect done(P), represents the use of press Pr on plate P;
- loadDepositBelt(R, A, D, P), with pre-conditions plate(P), robot(R), arm(A, R), loaded(A, P), depositBelt(D), empty(D), −failed(A), −failed(R) and −failed(D), and effects −loaded(A, P), empty(A), loaded(D, P) and −empty(D), represents the loading of plate P in the deposit belt D by arm A of robot R.

Within this problem, agent goals are very simple: the processing of any given plate P as soon as it enters the cell followed by its loading in the deposit belt once done. This is modeled by the following desires:

```
des(fzi,done(P),Tf,[0.8]) if bel(fzi,plate(P))
des(fzi,loaded(depositBelt,P),Tf,[0.9])
    if bel(fzi,done(P))
```

The placement of a new metal plate over the loading table is represented by the inclusion in the agents beliefs of the properties plate(plate1) and loaded(table, plate1). Such modification will trigger the agent's deliberation process. Within this process, the pre-condition of the desire to achieve done(P) is evaluated as true, turning such desire into an eligible one. In order to satisfy this desire, the agent's planning algorithm generates the following plan:

```
unloadTable(robot,arm1,table,plate1).
loadPress(robot,arm1,press1,plate1).
usePress(press1,plate1).
```

This plan proves, therefore, the possibility to satisfy the previously selected eligible desire, thus, it becomes a candidate desire. Such desire will then originate primary and relative intentions, leading the agent to execute the specified actions. During the process of executing the actions, it is possible for a fault to take place in one of the components, for instance, press number one (press1), which is denoted by the belief on property failed(press1). In this case, the action loadPress(robot,arm1,press1,plate1) becomes impossible due to one of its pre-conditions now being false. Once the agent notices such fault, it will have to re-plan its course of actions. Supposing that the agent has already executed the action unloadTable(robot,arm1,table,plate1), the new plan generated by the agent is:

```
loadPress(robot,arm1,press2,plate1).
usePress(press2,plate1).
```

As the agent was capable of generating a new plan to satisfy the initial desire, it remains a candidate desire, while the agent only had to modify its relative intentions in order to reflect its commitment to a different course of action. In this new course of action the agent will use press number two (press2) instead of number one to process the metal plate. If in the same situation the failed component were arm number one (arm1), denoted by the belief in the property failed (arm1) the achievement of the desire to process the metal plate becomes impossible, considering that the plate will be stuck in the defective arm. Many other combinations of faults in various cell components were tested, in which the agent's ability to try corrective actions was possible and where the fault prevented the agent to achieve its goals.

6. CONCLUSION

This paper has presented a new approach to inject faults in a system modeled using some type of fault-tolerant mechanism. This approach has been applied to a tool that allows to model a system as an agent that has the responsibility to plan the set of actions that have to be executed. This set of actions can, through the tools interface, effectively be executed and the results are reflected in the environment in which the agent is embedded to.

Using this tool, we could model several different types of faults, either environment faults or system faults, and inject these faults into the agent, which represents the system, or in the world view, which represents the environment. Throughout the fault injection we could visualize the behavior the system would have with the different types of fault-tolerant mechanisms used. For example, the re-planning performed by the agent is actually a form of forward error recovery; or n-version programming could be implemented using agent replication or using diverse plans to achieve the same solution (using a graph-based planner several different solution extractions are possible, i.e. several different plans).

Fault-tolerant systems modeled as autonomous agents allow the simulation of faults in terms of its interaction with its surrounding environment, which in turn allows the observation of system behavior under these conditions. This approach to system design might be an enabling technology for dependability testing. Throughout the use of fault injection, we could also detect situations where the agent had not been modeled properly and catastrophic failures happened.

Although this paper has shown the utility of using our approach to verify systems that use any type of fault-tolerant mechanism, we still have to apply this approach to real applications. The examples we have used so far are based on simulators, and some issues were not addressed, for example real time. Another addition currently being considered for the planning module is the usage of a constraint-based planning and anytime algorithms to augment the agent responsiveness in time-critical applications [16].

Acknowledgment: This work was partially financed by CNPq/Brazil and HP Brazil R&D.

7. REFERENCES

[1] Alferes, J. J., and Pereira, L. M. *Reasoning with Logic Programming.* Springer Verlag, 1996.

[2] Arlat, J. From experimental assessment of fault-tolerant systems to dependability benchmarking. In *IPDPS 2002* (2002), IEEE CS Press, pp. 135–136.

[3] Avižienis, A. A. *Software Fault Tolerance.* Wiley, 1995, ch. The Methodology of N-Version Programming, pp. 23–46.

[4] Bratman, M. E. *Intention, Plans and Practical Reason.* Harvard Press, Cambridge, MA, 1987.

[5] Coddington, A. M., and Luck, M. A motivation-based planning and execution framework. *International Journal on Artificial Intelligence Tools. 10*, 1 (2004), 5–25.

[6] Cohen, P. R., and Levesque, H. J. Intention is choice with commitment. *Artificial Intelligence 42*, 2-3 (1990), 213–261.

[7] Dastani, M., Hulstijn, J., and Meyer, J.-J. C. Issues in multiagent system development. In *AAMAS* (2004), ACM Press.

[8] der Hoek, W. V., and Wooldridge, M. Towards a logic of rational agency. *Logic Journal of the IGPL 11*, 2 (March 2003), 133–157.

[9] D'Inverno, M., Luck, M., Georgeff, M., Kinny, D., and Wooldridge, M. The dMARS architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems 9*, 1-2 (2004), 5–53.

[10] Fikes, R., and Nilsson, N. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*, 3-4 (1971), 189–208.

[11] Guessoum, Z., Briot, J. P., Charpentier, S., Marin, O., and Sens, P. A fault-tolerant multi-agent framework. In *AAMAS* (2002), ACM Press, pp. 672–673.

[12] Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. Fault injection techniques and tools. *IEEE Computer 30*, 4 (1997), 75–82.

[13] Jennings, N. R. On agent-based software engineering. *Artificial Intelligence 117*, 2 (2000), 277–296.

[14] Kumar, S., and Cohen, P. R. Towards a fault-tolerant multi-agent system architecture. In *AGENTS* (2000), ACM Press, pp. 459–466.

[15] Meneguzzi, F. R., Zorzo, A. F., and Móra, M. D. C. Propositional planning in BDI agents. In *SAC* (2004), pp. 58–63.

[16] Nareyek, A. Beyond the plan-length criterion. In *LNAI*, vol. 2148. Springer Verlag, 2001, pp. 55–78.

[17] Randell, B. System structure for software fault tolerance. *IEEE Trans. on Software Engineering 1*, 2 (1975), 220–232.

[18] Schut, M., and Wooldridge, M. The control of reasoning in resource-bounded agents. *The Knowledge Engineering Review 16*, 3 (2001).

[19] Wegner, P. Why interaction is more powerful than algorithms. *Comms. ACM 40*, 5 (1997), 80–91.

[20] Winikoff, M., Padgham, L., Harland, J., and Thangarajah, J. Declarative & Procedural Goals in Intelligent Agent Systems. In *KR* (2002).

[21] Wooldridge, M. *Reasoning about Rational Agents.* The MIT Press, 2000.

[22] Xu, J., et al. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Trans. on Computers 51*, 2 (2002), 164–179.

[23] Zorzo, A. F., and Stroud, R. J. A distributed object-oriented framework for dependable multiparty interactions. In *OOPSLA* (1999), ACM Press, pp. 435–446.