Validation of a Multi-Agent Architecture for Planning and Execution

Maria D. R-Moreno[†], Guillaume Brat[‡], Nicola Muscettola[*] and David Rijsman[‡]

[†] Dpto. Automática, Universidad de Alcalá, Madrid, Spain.
mdolores@aut.uah.es

[‡] MCT, NASA Ames Research Center. CA 94035, USA.
brat@email.arc.nasa.gov

[*] Lockheed Martin. CA 94055, USA.
nicola.muscettola@lmco.com

Abstract

Traditional autonomous architectures are composed of technologically diverse functional layers operating at different levels of abstraction which makes the integration and validation difficult. In this paper we present the validation procedure followed in IDEA (Intelligent Distributed Execution Architecture).

IDEA is based on Multi-Agent system, where each agent relies on a domain model, a plan database, a plan runner, and a reactive planner. We have integrated a simulator agent into the architecture to validate the whole system behaviour. The simulator agent shares the physical agent model but extended to consider all the possible cases that can occur in the physical layer.

We also present how model checking techniques can be used to explore the space of input scenarios to validate the reactive planner with the simulator agent. The space of possible failures and reactions is explored and analyzed to define a set of scenarios that trigger realistic uses of the reactive planner. Our goal is to provide a good coverage of representative reactions to off-nominal behaviours of the environment being controlled.

Introduction

Autonomous control software uses models to reason about the system that it controls and the environment it is in. It accomplishes a set of goals during a period of time, and it is able to reason about failures with little or no human supervision. Given the initial state and external goals, it generates a set of synchronized low-level activities that, once executed, will achieve the goals. If any action is not executed as expected, the system is able to recover in order to achieve the pre-defined goals.

So far, the use of autonomous control software has been limited to controlling research rovers and some satellites such as EO-1. The biggest obstacles to its deployment in high-profile missions are its lack of real experience (despite the DS-1 experiment) and the apparent difficulty in validating it. Testing effective and robust control strategies requires dealing with any path of execution in any component and any module. In traditional systems, this already implies searching through a huge state space. Because of their flexibility and their reactivity to unpredictable events, autonomous systems generate even bigger state spaces; and, unpredictability makes mission managers nervous. Yet there is no denying that autonomous control software could bring greater flexibility, and most probably result in greater science returns than traditional control software. So what should we do? The answer is simple: we need to design architectures for autonomous control software with validation in mind.

In this paper we present how the IDEA Agent-based architecture allows a whole system integration and model verification. The domain model shared by the planners describes the operational mode of the system. We decided to validate this system by developing a simulator agent which is also controlled by a planner. The domain model of this planner consists of the domain model in the IDEA system and a set of constraints describing how the environment can react to the commands generated. This set of constraints includes success criteria for the commands (which models nominal operational scenarios for the underlying system) and failure modes (which models off-nominal scenarios). This simulator interacts with the planners in IDEA to simulate the reactions of the underlying system and the environment in which it will operate. Model checking techniques are used to explore the space of input scenarios to validate the reactive planner with the simulator agent.

The paper is structured as follows. The first Section provides an overview of the architecture for Planning and Execution. Then, we describe in detail the structure and behaviour of the Agent Simulator. Next, we revise the Model Checking Techniques integrated in the Agent Simulator. After, we show a simple example of the approach. Finally, we present the works related to our approach and the conclusions and future works are outlined.

IDEA Architecture

IDEA is a model-based autonomy architecture, being the agent the core of the architecture and the Reactive Planner the control engine of IDEA.

Figure 1 shows the different elements that compose the architecture. For a more detailed description of the architecture, the reader can refer to (Aschwanden *et al.* 2006), (Finzi *et al.* 2004), (Muscettola *et al.* 2002).

- The Communication Relay: transmits data and events to other agents or systems.
- The Agent Relay: is responsible for responding to internally and externally generated events within an IDEA agent. It relays outgoing messages to Communication Relays and incoming messages to Reactors.
- The Reactive Planner: is the responsible to respond to internally and externally-generated events, data and the passage of time. As the default Reactor, the Reactive Planner receives all incoming messages not explicitly modeled to be associated with another Reactor.
- The Agent Timing Service: provides a clock to synchronize IDEA agents.
- The Plan Service Layer: is the module that ensures complete consistent synchronization of the execution context between the Agent Relay and the internal state of the Reactive Planner. The (PSL) database stores and updates the past, present and future state of objects in the domain.
- The Goal Loader: is designed to import externally-defined goals (a temporally flexible plan) into the PSL.

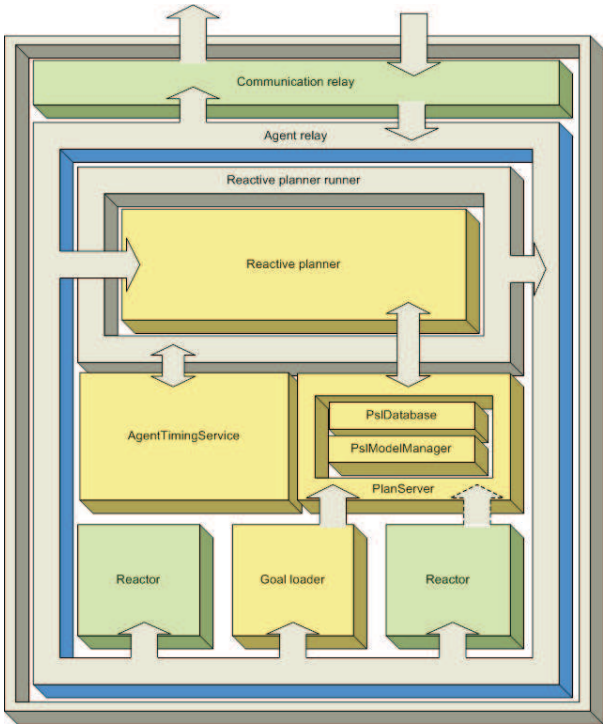

Figure 1: IDEA Architecture Overview.

At the heart of each IDEA agent is a declarative model describing the system that it can control, the activation of deliberative planning and the interactions with other agents.

IDEA models are declarative descriptions of legal operations of a system encoded in an XML based language *XIDDL* (XML IDEA Domain Definition Language). It allows us to describe interacting states and activities that can occur in complex systems in the same way as its predecessor DDL (Muscettola 1994). The main elements are:

- Class: is defined as a set of Timelines.
- Timeline (TL): is a logical structure used to represent states over time. A state is captured in a token. We can distinguish 3 modes of TLs:
  - Internal mode: describes the internal state of the controlled subsystem. State transitions are not communicated outside the agent and they are the result of compatibilities.
  - Goal mode: it represents a system which exerts control over the agent itself. State transitions can be the result of internal compatibilities or can be received from external systems. The agent will communicate return arguments and status parameters to the outside world (see parameters below).
  - Executable mode: describes the state that will be communicated to the outside world when the state transition is at the current time. This is also refered to as *dispatching* a command to the external system. The commands can be open or closed loop depending on the parameters defined in the model for the token. A closed loop command is a command for which the dispatching system expects a status back.
- Tokens: define the possible values that TLs can have. They can have zero or more parameters of the following types. Note that the the semantics of this type are applied for *executable* TLs. However, the semantics are inverted if these parameters are *goal* TLs.
  - `call_args`: this type of parameters are passed to the external system. They are expected to be determined before the command is sent to the external system.
  - `internal_modes`: this parameter is used for internal reasoning and not to communicate out.
  - `return_args` and `return_status`: these values are returned by the external subsystem. That is, the dispatching agent is expecting feedback from the external system. We can model as many `return_args` as we need but we can only define one `return_status` per token. The `return_status` parameter requires the definition of a boolean variable that is used internally by the Agent. If the value is *True* it means that a status

from the exterior is expected to be received indicating the command is closed loop. If the value is *False* it means that we do not expect a status, also referred to as open loop. The agent will enforce the values of these boolean flags during execution, that is if a status is actually received the flag is forced to be *True*, if the external system does not return a status at the time the dispatching agent expects it the flag is forced to be *False*.

- Constraints: temporal and parameter restrictions between tokens.

The tokens are executed only after they have appeared in a plan maintained in a central database. This can happen either because a controlling agent has communicated new goals or because some internal planner (reactive or deliberative) has generated appropriate subgoals. To be considered for execution, a token must allocated on an appropriate TL.

The Simulator Agent

In order to verify and detect inconsistencies in the model and verify the whole architecture, we use another IDEA agent with a mirror model of the original model. The goal is to guarantee that the original model can handle most paths of execution of the external systems. Because generating all possible paths it may be infinite in complex models.

For that, we need to simulate different scenarios to cover the possible situations that can occur. Having in mind that the dispatching system - the Agent - sends out commands and can require return values or status back from the external system, and that the Agent can receive commands to execute a goal, we can use this mechanism to simulate any kind of behaviour including the functional layer systems.

For building an agent simulator we need to define the model that allows us to define its behaviour. The simulator model will be based on the original model such that the assumptions of the external systems captured in the compatibilities are also in the simulator model.

Figure 2 shows the different elements in the model that compose the agent and the simulator agent. The box on the left represents the domain model that drives the agent behavior. It contains the Physical Model, that is the objects, TLs, tokens and compatibilities that define the system we want to model, and the non-Physical Model can contain the TLs, objects, tokens and compatibilities not directly related to external systems such as activating deliberative planning or load goals. The box on the right represents the agent simulator declaration. Both the agent and the simulator agent share the Physical model but the mode of the TLs is inverted. That is, a *goal mode* TL will be converted into an *executable mode* TL and the *executable mode* TL into a *goal mode* TL; *internal mode* TLs will remain the same as they are intrinsic to each Agent. The TLs in the simulator side are "mirrors" of the TLs in the agent side.

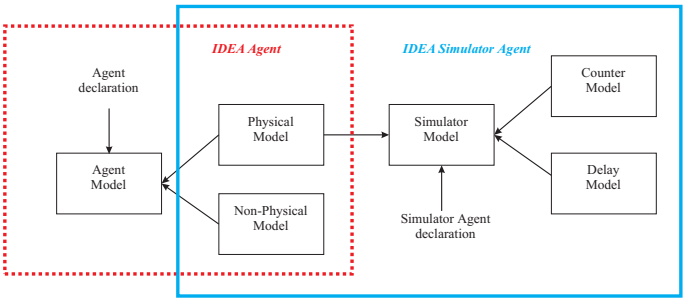

Figure 2: Agent and Agent simulator model structure.

The Delay Model refers to the gaps that will appear due to delays in the communication channel in each TL.

The Counter Model is the heritage from the simulation files that are used. Before the implementation of the Simulator Agent, the tests were created by hand using a non very intuitive syntax based on numbers and name tokens. For example, if there is a token called GPS with three arguments that refer to the x, y, and z position of the rover; and we want that after 2 time units the values of those parameters (i.e 3.14, 23.14, 13.14) are sent to the agent, the syntax used is as follows:

```
GPS    4  3  1
3.14   2  0
23.14  2  1
13.14  2  2
2      OK
```

The first row represents the token name, the ocurrence of the token in the plan (that is, the forth time the token appear in the database, will have the values set in the example), the number of parameters and an integer representing if we have or not a return status (0 means no, 1 means yes). Note that the index of the parameters starts by zero (last column in each parameter). The second column represents the time units when those values are sent. Finally, the last row represents when the return status value is sent and its value. Then, introducing errors in the creation of these tests is quite easy because the consistency of the values respect to the model are not checked.

So the Counter Model in Figure 2 will count the number of times that each token appears in each TL (second number in the first row of the simulation file) so we can have control over the commands we receive. The behaviour of the agent simulator can be split in two categories.

- Responding to dispatched commands: sending back the values of parameters and status of each

dispatched command in case `return_args` and `return_status` have been defined in the model for a token on a *executable* TL. For example, in the real rover model we are working on, there is a token called *Drive* that belongs to the *executable* TL *DrivingQueryTL*. This token has 1 `call_args`, *DriveTo*, 1 `return_args` that refers to the location it drove to, *DroveTo*, and 1 `return_status` parameter indicating if the external system considered the command to be executed successfully. Its possible values are *Success* or *Fail*.

The agent simulator has to decide how to respond each time it receives a *Drive* goal. For example it receives a *Drive* goal with call parameter value *ROCK1* for *DriveTo*. The simulator agent now has to decide when to send back a status, what value for the status and what value for *DroveTo*. Based upon the model the simulator should know what the allowed values are for these parameters given the context of the *Drive*.

The time and the values will be selected using model checking techniques as explained in the next section. To do this we need to keep track of how many times we receive a command. We do this by enhancing the Physical Model with the Counter Model.

- Dispatching commands to the agent: the simulator is responsible for deciding the timing of the commands, what commands and what the values are for the call parameters. For the receiving agent these commands are goals and once received it has to find a plan consistent with the new goals.

Figure 3 shows the integration in the Simulator Agent of the counter and delay models for an *executable* TL in an agent. The *Tk1* token in the Agent Side sends its start time to the mirror TL in the Simulator Side which receives it with a delay. The same process occurs when the token finishes. Then, the *meets* compatibility triggers the start of the *Tk2* token. Here the situation varies because of the status value. The Counter TL in the Simulator side (defined as an *Internal* TL) checks the occurrence of the token in the TL and sends the value(s) back to the Agent side. Let us say that in the Agent side there is a compatibility to meet the token *Tk3* if the status is for example Fail. In the Simulator Side no decisions should be made until the next start message is received (token *Tk3*). Then a *Buffer* token is added in both counter and mirror TLs to fill the gap due to the delay in receiving the next start time token.

Then, we have built the appropiate environment to generate the tests compatible with the model. The next step is to decide: what do we want to test, and what values do we want to set. Next section describe how this could be done.

Model Checking Techniques

This section describes how we generate scenarios for the simulator agent. On one hand, simulating nominal behaviour is a simple matter of going through combinations of commands. On the other hand, off-nominal is trickier since it involves guessing what environmental conditions or mishaps in the underlying physical system can trigger reactive planning. Our approach consists of modelling the space of off-nominal actions and using model checking to generate a representative set of off-nominal scenarios.

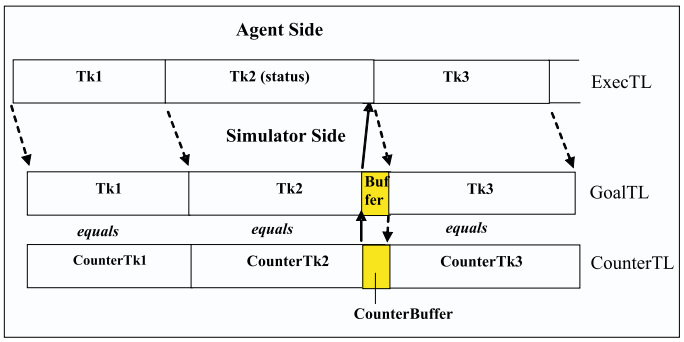

Figure 3: Counter and Delay Models design.

Model-based test generation

Model checking is a method to verify finite state systems formally. This is achieved by verifying if the model (derived from the design phase or by abstraction of the code) satisfies a logical property (derived from the requirements). Properties are often expressed as temporal logic formulas, but simple assertions can also be checked.

The model is usually expressed as a directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node. The nodes represents states of a program, the edges represent possible executions which alters the state, while the atomic propositions represent the basic properties that hold at a point of execution. The problem can be expressed mathematically as: given a temporal logic formula p and a model M with initial state s, decide if:

$$< M, s > \models < p > .$$

In general, model checking explores systematically a state space while examining all possible choices at decision points. However, exhaustive search is often impossible to achieve. Fortunately, when properties are violated by the system (under exploration), the model checker can be content with simply returning a trace illustrating the violation (also called, a counter-example). This is very similar to the process of planning, during which a planner looks for and stops as soon as it finds a plan satisfying the planning constraints.

The process of creating counter-examples in model checking can be exploited to generate test cases guaranteeing a certain coverage for the system under test.

The desired coverage criteria are gathered in a property (which can be as simple as a conjunction of assertions on the values of state variables) so that the violation of the property will cause the model checker to return a (counter-example) path that satisfies the coverage criteria. A test case can then be extracted from the trace by identifying the input values (and possibly their order and timing). This process may look redundant since the model checker is exploring the path that we want the test to exercise. However, meeting the coverage criteria (i.e., violating the property representing the criteria) does not necessarily imply a full path exploration. In this exercise, the model checker is really looking for a quick way to meet the criteria. This can actually be a problem in the sense that modern model checkers are getting too good at returning minimal counter-examples to illustrate violations. In our case, it means that we generate test cases that exercise very little of the system. This problem can be solved by allowing the model checker to start its exploration from arbitrary points in the state space; this gives us a sort of "bunny hopping" strategy to test generation.

SAL

SAL (Symbolic Analysis Laboratory) is a framework for combining different formal method tools to calculate properties of concurrent systems. The goal of SAL is to be scalable, automatic, and cost effective. SAL is really an intermediary language, developed conjointly by SRI, Stanford, Berkeley, and Verimag, which can be used to integrate various tools based on formal methods. In this work, we are using SAL 2 developed by SRI (de Moura *et al.* 2004).

For example, the contribution of SRI to the SAL framework deals with augmenting the PVS theorem proving framework with tools for abstraction, invariant generation, program analysis (such as slicing), theorem proving, and model checking. In particular, they use PVS to reduce a verification problem into a finite form and then model checking to calculate verification properties. SRI also integrates a bounded model checker for timed automata into the SAL framework; it gives them the ability to deal with models, or systems, constrained with time information. The current version of SAL also includes an infinite-bounded model checker which provides bounded model checking for systems defined over infinite data types, such as integers and reals. SAL can deal with LTL as well as CTL properties. SAL describes transition systems in terms of initialization and transition commands. These can be given by variable-wise definitions in the style of SMV or as guarded commands in the style of Murphi.

The SATELLITE Domain Example

In order to validate the ideas mentioned above, we have started by modeling in XIDDL a simple domain, the SATELLITE domain where our ideas have been tested. This domain belongs to the set of domains used in the 3rd International Planning Competition (IPC'02). Given a collection of satellites that contain different instruments in specific modes, the goal is to take pictures in some directions using the available instruments in the satellite. Let us resume the actions that belong to this domain:

- `Turn_to`(?s - satellite ?d_new - direction ?d_prev - direction): after the action is performed, the satellite points to a new direction.
- `Switch_on`(?i - instrument ?s - satellite): at the end of the action, the power is available in a specific intrument inside the satellite.
- `Switch_off`(?i - instrument ?s - satellite): is the opposite of Switch_on, that is, after the action is executed, the power in the instrument is off.
- `Calibrate`(?s - satellite ?i - instrument ?d - direction): once the satellite points to the target direction and the instrument is switched on, then the instrument is calibrated and ready to take an image.
- `Take_image`(?s - satellite ?d - direction ?i - instrument ?m - mode): if the instrument is calibrated and switched on, then the image is taken.

Figure 4 shows the Timelines (TL) considered for the SATELLITE domain. Basically, we have considered 4 TLs, each one representing the evolution in time of a physical component. In this case, we have abstracted the satellite, the power, the instrument, and the mode. These four TLs are considered as *executable* TLs since the tokens of the TLs can send messages to the outside world and may receive return and status values (see section to an explanation of the type of parameters tokens can have).

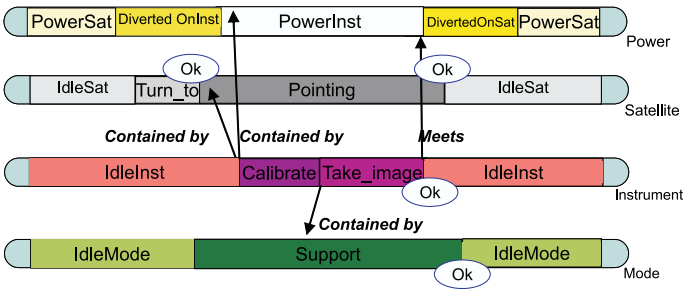

Figure 4: The SATELLITE domain using the Timeline representation.

For the purpose of the example, we are going to consider that only the `Turn_to`, `Pointing`, `Take_image` and `Support` tokens receive a status from the external world, which in our simulating environment, that value is generated by the simulator agent.

Since there cannot be gaps between tokens, we have added extra `Idle` tokens to all of the TLs. In the case of the Power TL, the `PowerSat` represents the Idle token

and instead of the `Switch_on` and `Switch_off` operators, we have modelled the power status using the `DivertedOn (Sat/Inst)` and `PowerInst` tokens.

Tokens in the same TL are connected by the *meet* compatibility. The relationship with other TLs is defined using any other temporal relation. For example, the `Calibrate` token has to be contained by the `PowerInst` and `Pointing` tokens.

We now describe the model we used for the simulator input generation. We want to generate scenarios that can exercise the off-nominal scenarios. Therefore, we concentrate on the off-nominal behaviour of the underlying system and the environment. Yet let us start by describing model variables related to nominal activities. Each activity (or command) can generate a:

- status, such as **OK** or **Failed**,
- occurrence index, which indicate the occurrence of similar activities

Now, the off-nominal scenarios can come from the following places:

- message delays: each command and status reply go over the network, and therefore, may be subject to delays. Arbitrarily, for the sake of this example, we decided that delays can range from 1 second to 5 seconds.
- latency: the beginning of an activity might not correspond to the frequency interval at which the agent is controlling the system. The latency range is given by the frequency of the control.
- send and receive times: times at which message are emmitted and received by the agent and the simulator. It is closely related to the message delay.

The transitions in our model are given by the nominal behavior, which is given by the compatibilities in the Model of the agent. For example, the activity **TurnTo** is met by the activity **Pointing** on the **Satellite** TL whereas the activity **Calibrate** makes sense only for the **Instrument** TL and is met by a **TakeImage** activity. Note that this model also includes timing information for each activity. Now, the off-nominal scenarios are described by a state machine, which takes into account timing information such as message delays and activity durations. Different time values lead to different **status** state. For example, for the activity **TurnTo** on the **Satellite** timeline, a duration of 5 seconds leads to an **OK** state (corresponding to the nominal behaviour) while a duration of 7 seconds leads to a **Failed** state (corresponding to an off-nominal behaviour which will trigger the reactive planner).

Now, this model really needs to be expressed using the SAL language. Then the *sal-atg* tool can be used to generate test inputs (Hamon *et al.* 2004) (Hamon *et al.* 2005). This is achieved by adding trap (boolean) variables in state of interest. These variables are initially set to *FALSE* and are set to *TRUE* when their corresponding state is explored by the model checker. Now, the states containing trap variables are chosen in such a way that we hit all boundary conditions in the timing variables, giving us a coverage of all the scenarios in which timing values cause the agent to miss its deadlines. This allows us to generate scenarios such as "At occurrence zero of the **TurnTo** activity a duration of 7 seconds will cause the simulator to return a **Failed** status".

Although we have presented a simple example where there is just one agent, the IDEA architecture allows us to create as many agents as we need and then, simulator agents. In the XIDDL domain description we also have to provide (see the concepts introduced in section 2), the agent topology and configuration description, consisting of: the *latency*, a declaration of all of the known initial subsystems associated to that agent and the communications *channels* between the agents and the subsystems in the domain. Since we want to simulate the subsystems behaviour, the target channel is modified by the name of the simulator agent in charge of its behaviour. In the same way, in the simulator agent description, we write as the target channel of that subsystem the name of the normal agent. Like that, the communication between agents and simulator agents is guarantied.

Related Work

The Remote Agent (RA) (Muscettola *et al.* 1998) was based on a three-layer architecture - Functional, Execution and Planning/Scheduling - each one using different technologies which complicated the integration and testing of the whole system. Another complication of the layered architectures is the lack of access from the Planner to the Functional Level. Most of the traditional autonomous and robot architectures follow this approach, differing in the degree of dominance of some layers over the others (Estlin *et al.* 1992), (Borrelly *et al.* 1998), (Estlin *et al.* 2000).

More recent approaches, as CLARAty (Volpe *et al.* 2001), try to overcome some of these drawbacks using a two-layer architecture: the Functional and the Deliberative layers. The Functional layer follows an object oriented design for the hardware components. The Deliberative layer integrates planning and execution through a tightly-coupled Database that allows synchronizing the same information using two different representations: the one from CASPER (planning) and the one from TD (execution). The simulation of the system consists of method calls that emulate some behaviour. In IDEA, the planning, execution control and simulation share the same modelling framework what makes the integration and validation an easier task. In the model, it is possible to abstract what to simulate and error injection can be easily incorporated.

Another two-layer architecture is the model-based approach pursued by (Williams *et al.* 2004). This framework consists of a Reactive Model-Based Programming Language (RMPL) and an executive (Titan). A model-based program specifies two inputs, a control

program and a system model. The language abstracts from the functional level so the programmer can reason in terms of state variables not directly corresponding to observable or controllable states, which is different from IDEA where the model is directly integrated with the functional level. From the simulation point of view, RMPL cannot provide time related utilities such as timeout definitions or warping because of the state variables representation. The simulation is done synchronously and backward from observations. Our architecture instead, provides any time related utilities and richer scenarios than just the ones from observations.

Conclusions and Future Work

In this paper we have presented an architecture for planning and execution based on Multi-Agent systems. An agent can communicate with multiple agents both controlling and controlled. Although two agents could mutually control each other. The allowed communications are governed by a Model that describes which procedures can be exchanged with which agents.

Within this framework we have integrated a Simulator Agent that can simulate some behaviour. The goal is to guarantee that the model can handle all paths of execution. For that we have simulated different scenarios to cover all the possible situations that can occur. Model checking techniques are used to explore the space of input scenarios to validate the reactive planner with the simulator agent.

The approach has been first tested in a simple satellite model from the IPC'02 competition. Then, in the real rover model Gromit (Finzi *et al.* 2004). Thanks to the simulator agent we have automated the simulation process (before the files were written by hand) and created tests that the testing group haven't thought about them.

For the future we also want to use co-evolution techniques to optimize the scenarios generated under some stress parameters, i.e. reactive time or time between deliberative and reactive planning. And we also want to study if probabilistic planning can be integrated in this architecture.

Acknowledgements

R-Moreno's work at NASA Ames has been supported by the Spanish Ministry of Education and Sciences. Nicola Muscettola's work was conducted while he was at NASA Ames. This work has been partially funded by the Castilla-La Mancha project PAI07-0054-4397 and the CICYT project ESP2005-07290-C02-02.

References

Pascal Aschwanden, Vijay Baskaran, Sara Bernardini, Chuck Fry, M.D. R-Moreno, Nicola Muscettola, Chris Plaunt, David Rijsman, and Paul Tompkins. Model-Unified Planning and Execution for Distributed Autonomous System Control. In *AAAI 2006 Fall Symposia. Washington DC (USA), October*, 2006.

J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The ORCAD Architecture. *International Journal of Robotics Research.*, 17(4), 1998.

L. de Moura *et al.* SAL 2. In *Procs. of Computer-Aided Verification, CAV 2004*, 2004.

T. Estlin, G. Rabideau, D. Mutz, and S. Chien. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *The Journal of Real-Time Systems,*, 4(1):37–53, 1992.

T. Estlin, G. Rabideau, D. Mutz, and S. Chien. Using Continuous Planning Techniques to Coordinate Multiple Rovers. *Elec. Trans. on AI*, 4:45–57, 2000.

A. Finzi, F. Ingrand, and N. Muscettola. Model-based Executive Control through Reactive Planning for Autonomous Rovers. In *Procs. of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan*, 2004.

G. Hamon, L. de Moura, and J. Rushby. Generating Efficient Test Sets with a Model Checker. In *Procs. of SEFM'04*, 2004.

G. Hamon, L. de Moura, and J. Rushby. Automated Test Generation with SAL. Technical Report CSL Technical Note, SRI, USA, 2005.

N. Muscettola, P. Nayak, B. Pell, , and B.C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, 1998.

N. Muscettola, G.A. Dorais, C. Fry, R. Levinson, and C. Plaunt. IDEA: Planning at the Core of Autonomous Reactive Agents. In *Procs. of the Workshop On-line Planning and Scheduling, AIPS 2002, Toulouse, France*, pages 49–55, 2002.

N. Muscettola. HSTS: Integrating Planning and Scheduling. In M. Zweben and M.S. Fox, editors, *Intelligent Scheduling*, pages 169–212. Morgan Kaufman, 1994.

R. Volpe, I.A.D. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty Architecture for Robotic Autonomy. In *Proceedings of the 2001 IEEE Aerospace Conference*, 2001.

B.C. Williams, M. Ingham, S. Chung, P. Elliott, and M. Hofbaur. Model-based programming of fault-aware systems. *AI Magazine*, 24(4):61–75, 2004.