DyVE: Reactive Software Validation Through A Dynamic Validation Tool

Christophe Arlen*
Thomson Airsys
Multidomain Control Centers
7, rue des Mathurins
F-92223 Bagneux Cedex
FRANCE
+33 1 40 84 18 33
arlen@sdc.thomson.fr

Bruno Berstel**
Ilog S.A.
9, rue de Verdun
B.P. 85
F-94253 Gentilly Cedex
FRANCE
+33 1 49 08 35 00
berstel@ilog.fr

April 24, 1996

Abstract

In the context of its activities in Air Traffic Control, Thomson Airsys developed DyVE, a workshop for developing environments of validation through simulation. This workshop uses the language ILOG TALK to integrate C++ components and to make those components dynamic—the key to the resulting validation. This article describes the DyVE workshop as well as this dynamic quality, the decisive element in the validation process.

1 Introduction

DyVE (Dynamic Validation Environment) [8] is a workshop for developing validation environments for systems through simulation. The workshop itself was developed between 1994 and 1995 by the NAOS team at Thomson Airsys. The project NAOS (New Air Traffic Management Open Systems) operates within the Technical and Scientific Direction of Thomson Airsys; its purpose is to study new computer technologies and their applications to future Air Traffic Control systems.

The studies and demonstrations coming out of the NAOS project required the development of a series of simulators for air control functions (such functions as managing flight plans, synthesizing radar paths, coordinating control centers, controlling aircraft, etc.). Those simulators were needed within demonstrations where the goal was to validate the application of a set of techniques (such techniques as C++, rule-based programming, distributed programming, shared objects, etc.) to Air Traffic Control functions.

2 Validation through simulation

The need to validate a software system with respect to its specifications arises at various stages during its development. Systems-engineering techniques and tools (such as RDD 100 or BONES) assume an *a priori* validation of the software system. Tests are, however, *a posteriori* validation; this is the kind of validation addressed by DyVE.

Validation tests occur regularly and frequently, whether tests of a simple function, tests of a module, tests of an operationally significant functionality, right up to integration tests where interactions between elements of the software system are verified. Although commonly recognized as unavoidable, tests have a cost—often underestimated. We shall examine here the reasons for this cost and the way a tool for reactive validation such as DyVE can reduce this cost, thus bringing back frequent tests into the development cycle.

*DyVE Product Manager

**ILOG TALK Software Engineer, former member of DyVE team

At each level, *a posteriori* tests validate the operability of a software element with its adjacent elements. For that purpose, those tests require the presence of those adjacent elements and preferably in their validated state. In many cases, testers have to resort to solutions from simulated functions adjacent to the element to validate. They will do so to work around the fact that those adjacent elements are not yet available or still too rigid. One may devote considerable effort to those simulation solutions; the benefits one expects to get from them justify it; but these efforts generally do not go as far as producing highly refined nor greatly varied simulations.

In contrast, using a framework of validation through simulation lets us amortize the validation effort made at each step in development. In fact, that is the way the NAOS project works: we implemented a test-bed that makes it possible to synthesize a simulator quite rapidly to validate a software element incrementally.

Getting results is the primary aim of such a test-bed: producing simulations useful in the validation is the point, afterall, even more important than achieving irrefutable realism. The usefulness of the test-bed will also derive equally from the set of validation services independent of the simulation model. We detail these services later in this article, but here we mention, as one example, the service of controlling time (breaks, speed, etc.).

Moreover, this test-bed will be useful only if the effort required to adapt it stays focused on the application realm of the software system that needs to be validated. Other considerations, such as the interface among simulated elements themselves or between them and the one to validate, must exploit, indeed capitalize on, the expertise of the framework, to relieve the users from that concern. Likewise, designing the model for the functions adjacent to the element to validate has to be made easier by reuse of existing efforts and/or by taking advantage of a rapid development environment.

Finally, using a simulation environment has to be simple in order to stay competitive with the costs allowed for validation phases. It must put a tool for preparing scenarios at the disposal of the user, and it also has to provide controlled environment for supervising how the simulation proceeds.

Those were the constraints that DyVE had to respond to by trying to achieve modularity and genericity in the services that it provided and by introducing maximally dynamic qualities at every level. Adopting object orientation and using a dynamic object language, like ILOG Talk, in this task made the decisive difference.

3 The tools used

DyVE is written in C++ and in ILOG Talk.

ILOG Talk [5, 6] is a Lisp dialect. As major Lisps (esp. Common Lisp), it comes with a compiler, an interpreter, an automatic memory management (GC), a meta-object protocol (MOP), high-level libraries. Its specificities as a Lisp system (esp. in contrast with Common Lisp) is that it is modular, oriented toward application delivery, Posix compliant, small, that it compiles to C, has a straightforward C++ interface mechanism, is shared library based, and that its runtime libraries are free.

Talk was initially used in DyVE as the language for writing scenarios. Then benefits were taken from its high-level libraries and development environment [3], and now the tool layer in DyVE is written in Talk. Python [1] was taken into consideration as a scripting language; ILOG Talk was prefered because of its robustness as a programming language and its ability to automatically generate interfaces (known as *bindings*) to C++ libraries [2].

The other ILOG products used around DyVE are ILOG Views and ILOG Rules. ILOG Views [7] is a C++ Graphical User Interface running on Unix and Windows platforms. As a C++ library, it has a Talk binding, thus enabling dynamic graphical interface programming. ILOG Rules [4] is a C++ library and a preprocessor used to write expert systems integrated in C++ programs.

4 The World According to DyVE

Each software element to validate is integrated into a software system which is in turn conceptually divided into entities. Each of those entities serves as the model for one operational functionality. The closer these functionalities are to the element to validate, the tighter will be the correspondance between the functionalities of the model and reality.

In the validation simulator, each of the functionalities is implemented by an *agent*, serving as the model of the operational characteristics (the *expert layer*) and collaborating with other agents, the element to validate, and the user of the simulator. An agent exploits a range of services (the *tool layer*). There is a clean and clear separation between these two layers. In fact, the tool layer must be independent of the operational realm of the simulator. The expert layer focuses on the model of the functionality embodied in a given agent, providing the expertise of the domain, whereas the tool layer has to supply the power and flexibility demanded by validation tools.

Globally, the software system includes the process or processes implementing the element to validate plus the agents that simulate the connected functionalities. These agents are distributed among one or more processes, and the processes, in turn, may be distributed among one or more machines in such a manner that the tool layer in no way constrains this distribution. Furthermore, this distribution must be transparent to the expert layer. Indeed, the way these parts are distributed is entirely up to the user, possibly dictated by the element to validate or by the middleware in use.

4.1 Expert layer: a simple example

DyVE has been used or is in use in several projects at Thomson Airsys, all of them related to Air Traffic Control. We present here (see fig. 1) a simplified example that gathers the typical aspects of a software system using DyVE to validate one of its elements.

The element to validate is an air traffic controller working position (CWP); it is connected to other CWPs, to an air traffic generator (ATG) from which it receives radar echoes, and to a flight plan processor (FPP) with which it exchanges flight plan information. The whole is the software system and would more realistically include models (agents) of adjacent control centers, several types of controller positions, and other processor connected to the CWPs, such as a meteo information provider, a conflict detection system, and so forth.

In the project that inspired this example, the air traffic generator was written in C++, and was divided into four agents; the flight plan processor was written in C++ and included expert systems generated by ILOG Rules; the CWPs adjacent to the position to validate were written part in C++ and part in ILOG Talk.

Scenarios are generated off-line by the means of a scenario preparation utility. They are read by the agents at the beginning of each validation session. During the session, the simulator and the agents are controlled by the managers operated by the validator. As we shall see, these managers enable the validator to dynamically modify or extend the scenario and the behaviour of the agents. The production of the agents (traces ranging from standard output) is used by the validator to test the element to validate.

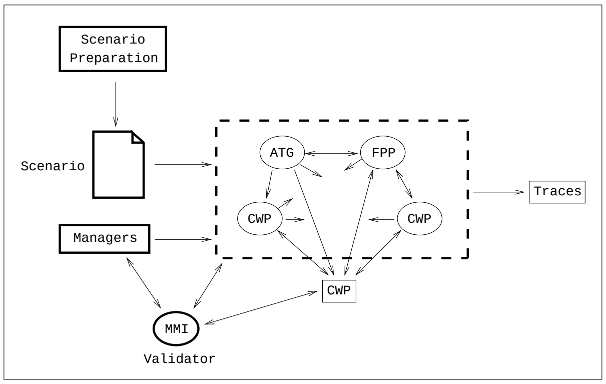

Figure 1: *A simplified example of the use of DyVE.*

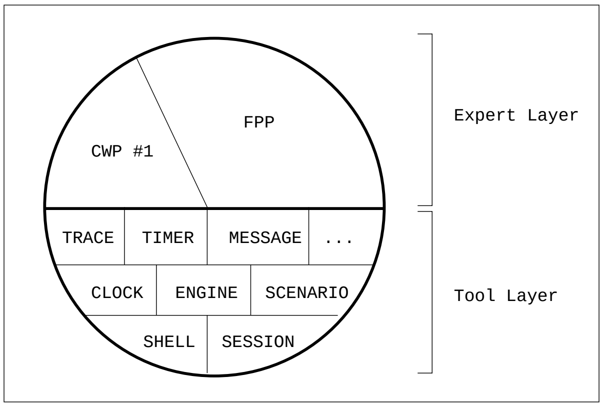

Figure 2: *A process containing agents.*

4.2 Tool layer: services

The tool layer (see fig. 2) offers a set of services to expert layers for modeling their functions and for integrating them into the test-bed. These services are implemented as distinct libraries, so each process loads only the services used by the agents within it. Moreover, the object oriented design of these libraries makes it easier to extend the tool layer simply by writing new services.

The *shell* and *session* services. These services—*shell* and *session*—are the basis of the tool layer. They implement the division of the software system: i) between simulated elements (agents) and external elements (real-world or those to validate); ii) also through the distribution of these elements across processes and machines. Additionally, they guarantee that the libraries that implement the agents are loaded appropriately for the architecture chosen for the current validation session and thus insure the services used by these agents are loaded, too.

The *scenario* service. The *scenario* service enables the agents to load working data they might need (as explained in Section 5). It also guarantees them that initializations and resets will be executed appropriately at the beginning and end of the scenario.

The *engine* service. The expert layer provides the model of operational functionalities for agents on the basis of a reactive behavior model. What each agent does is designed as reactions to stimuli. These stimuli can be perceived directly by the expert layer of the agent; such a stimulus might be, for example, the action of a user in the possible human-machine interface of the agent. The stimuli can also reach an agent through the tool layer; they would thus be delivered in the universal form of *events*, such as the arrival of a message, the expiration of a timer, and so forth.

When an event occurs for an agent, the event is submitted to the *rule base* for that agent. These rules select a set of actions to carry out, that is, the treatments for the agent to launch. Managing the rule base of an agent, submitting events to the rule base, and launching the treatments by agents are all activities of the *engine* service.

The *message* service. The *message* service is used by agents to exchange operational messages. It offers aliasing for addresses thus allowing agents to be reused from the architecture of one simulator to another without any change in their code, independently of their real identity and of the physical location of their interlocutors. When this service receives a message, it synthesizes an event ("arrival of a message") which is then put into the hands of the *engine* service to be handled appropriately.

The *clock* and *timer* services. The *clock* service maintains the simulated time, an affine function of the real time. It can be regulated by the clock manager. (See clock manager in Section 4.3.) It can also provide agents with the current time.

The *timer* service is used by agents to schedule future actions. The *engine* service introduces rule bases; as a consequence of the use of these bases the timers can be exploited in a very natural way: the agent adds a rule to its rule base to specify the time at which the rule is "true"; at that specified time, the *engine* service starts the action scheduled that way.

The *trace* service. The *trace* service enables the agents to qualify the messages they output, in order for the validator to control them through the trace manager, described below. This service relieves the agent from the need to know what level a detail in the output will be of interest at runtime—a usually impossible guess. It is important that the designer of the agent can add as much traces as he or she wants, since the traces are the basic echo of the behavior of the software element to validate.

4.3 Validation services

Up to this point, we have described the services offered to designers of simulators to implement the expert layer. DyVE also provides services to the user of a simulator to help with validation. These services include a graphic tool for preparing scenarios; this graphic tool makes it possible to build scenarios in terms of the validation objectives. Other services include a group of *managers*, tools for the validator to control the simulation in order to get validation results or even to refine such results.

Session manager. The *session manager* lets the user first display the possibilities and then choose the architecture of the simulator for a validation session. Architecture in this context involves the kind and number of agents.

Scenario manager. During a validation session, the user can work with several scenarios in succession. The *scenario manager* lets a user choose a scenario, start it, and then end it.

Clock manager. A user exploits the *clock manager* to stop or to restart simulated time. The clock manager also determines the speed at which simulated time elapses. Additionally, it can set jumps, either jumps at regular intervals, jumps to a given time, or jumps to the expiration time of the next timer.

Trace manager. Agents emit trace output describing the progress of their work. This output can be controlled by the validator by the means of the *trace manager*. The control works on an agent basis and relates to the medium (file, standard output, terminal window…), the language, etc. used for the output, as well as the criteria of whether to emit the output or not.

Command manager. In Section 6, we will show how its dynamic aspect makes DyVE easier to control. Both actions fired by managers and other actions with no related manager (for the time being) are accessible through commands from the *command manager*. The command manager is nothing more than a conventional command line, but since it is in fact a manager, the overall homogenity is greater. Furthermore, the user can choose process by process whether or not to include the services of this manager, just as for other managers.

5 The scenario

DyVE uses scenarios, familiar from the simulation realm, to validate a software element. A scenario defines a situation in which we want to study the behavior of the element to validate, or the behavior of the entire software system. The scenario provides the data and stimuli that would come from the real environment in a non-simulated system; these data and stimuli feed models. This is the way the domain expert, the designer of the models that make up the simulator, sees things.

The programmer who designs the tool layer has a symmetric view. Pushed to its extreme, this symmetric way of looking at things assumes that DyVE provides no more than the test-bed, and that everything else (that is, the set of expert layers of the agents) is just scenario.

The person responsible for validating the system will have the outlook of a user of the validation environment based on DyVE: the simulator, tool layer and expert layer united, are all there to provide greater ease at work. This person expresses himself or herself through the scenario. He or she wants to get the most out of it in order to do useful work. Accordingly, a scenario is not only a source of data for agents; it must also be the place where parameters for these agents are set. And while we are at it, why not set parameters here for the services of the tool layer so that we can submit the element to validate to more varied or even unexpected conditions. By using a scenario to set parameters for agents, we avoid making the validator oblige the development team to intervene again in the agents' code—a practice that would greatly prolong the response time and overburden the models. However, the fact of setting parameters for agents in a scenario should not restrict the validator to setting numeric values; he or she has to be able to set parameters for the models that specify behavior.

At assessment time, everyone agrees to give the scenario a central place in the simulator. Likewise, for formulating scenarios, everyone demands an open language, capable of expressing complicated ideas and even behavior. In practice, a scenario language should be independent of the operational domain of each simulator, since we decline to invent a new language for every system we have to validate.

An interpreter corresponds to the language of scenarios, an interpreter located in the tool layer. Yet it should be possible in the scenario to manipulate data structures introduced by agents in order to specify there the behavior used by these agents. For that reason, the interpreter must offer a language connection with the agents.

With respect to the implementation of scenario services, the DyVE team concluded that it was not practical for them to design such a language and then implement the interpreter for it. Clearly, it would be more economical to choose an existing language already on the market to get better results than one could expect working alone. After a careful study of current offerings, the DyVE team chose the language ILOG Talk. In the next section, we will cover the advantages of this choice, but for now, we should mention these points:

- The robustness, completeness, and power of the language insures a good basis for future development. In effect, we need to guarantee that the language we choose for scenarios will not impede our implementation of the central role of scenarios.

- The strong and easy connection with C++ makes it possible to integrate the models and to reuse data structures and modules that we have already developed, and there was no question of our throwing them away!

- There is also the possibility of using ILOG Talk in the tasks of rapidly prototyping agents, as well as in efficiently programming the tool layer. These observations naturally lead to using ILOG Talk for more that we initially targeted it.

- Finally, ILOG Talk can be interpreted and/or compiled into the machine native format (`.o` files); this offers portability, flexibility and efficiency. Likewise, its binding with the GUI tool ILOG Views is advantageous.

6 What does a dynamic object language offer?

6.1 Implementing the scenario

ILOG TALK is the language in which the scenario is expressed. The phrases in the scenario that provide data useful to the simulator are consequently TALK expressions; those expressions lead finally to function calls. These functions, called in TALK in the scenario, are implemented in the agents. There they can be in TALK or in C++, thanks to the binding mechanism offered by ILOG TALK. This binding mechanism automatically furnishes a TALK interface for a set of C++ classes and methods. Thus by systematically binding the C++ code of the agents, we can manipulate all the data structures introduced by the agents, and we can do so from the scenario; we can also have the agents implement the functions called in the scenario, and do so in C++ without imposing any constraints on the expert layer.

Since we use TALK to formulate the scenario, we avoid writing a parser, an interpreter, and so forth, for the scenario. Nevertheless, the scenario remains independent of the domain and open to new agents. It leaves great latitude to the validator about the data he or she might introduce in the scenario. Because there is C++ binding in TALK, and since we use it, this latitude does not impose any constraints on the designer of the models; he or she can reuse existing C++ code, even opening it to use from a scenario.

In short, the scenario itself stays on the expert level although all the mechanisms around it are on the tool level. Thus we have been able to separate those worlds, so they offer the maximum power for our work.

6.2 Implementing commands

A user of the simulator operates on agents and exploits the services of the tool layer through commands. Here again, we use ILOG TALK, this time to formulate the commands. A command is a TALK expression, resulting finally in function calls.

The functions called in TALK by commands could be implemented in TALK, for example if the agents or services to which they are destined are also written in TALK. By the same C++ binding mechanism—the one we use for the scenario—commands could equally well be written in C++, particularly if their destinations are already written in C++.

Just as for the phrases of the scenario, for commands, we take advantage of the complete openness of agents, including the fact that the same data structures they manipulate can be handled in commands as well. Symmetrically, the command mechanism will be independent of the operational domain. In consequence, we can take into account new functions or even new domains without extra cost.

The user generates commands, directed to agents or services, by means of the TALK command line embodied in the command manager or by means of graphic interfaces; those graphic interfaces involve other managers or windows incorporated with agents. Since the ILOG TALK development environment (where the command manager takes over the command line) is integrated with EMACS, in DyVE, we thus have a validation tool integrated with the development environment—a considerable factor in productivity.

6.3 Homogeneity among the interpreters

As we mentioned, scenarios and commands share the same language: ILOG TALK. Since their language is the same, commands can be used within scenarios. Likewise, fragments of scenarios can appear in the command line. This commonality saves work for everyone: the tool layer will have only one interpreter to take into account; the functions of the agent will serve as scenario phrases and commands.

Inserting commands into the scenario becomes truly interesting when we use the form `at`, introduced by the *timer* service. From a scenario, it is thus possible to program a command for a given simulated time. Since it is feasible to send phrases of the scenario from the command line, and thus during a simulation, it is possible to modify or enrich the scenario dynamically—a very interesting possibility.

Since we chose for commands and scenarios to call functions implemented in the agents, it becomes possible to consider commands and scenario phrases as entry points in agents. In that way, we break through the customary rigidity of a scenario. We gain considerable flexibility and expressive power needed in a command language.

6.4 Dynamic actions of the validator

The validator makes use of these entry points to get into the exact operational situation of interest. To do so, he or she could launch treatments in agents by command to manipulate their internal data, since the validator uses the same data structures.

For example, in a software system serving as the model for an Air Traffic Control center, the scenario will certainly include a list of declared flight plans, which in turn lead to radar echoes, and so forth. To mention a flight plan in a scenario, we use TALK to call to a function (whether TALK or C++) named, for example, `add-flight` with arguments defining the characteristics of the flight by means of structures handled by the agents. As the scenario unwinds, then from the DyVE command line, we could create all the objects defining a new flight plan and call the function `add-flight` for them. In that way, we add a flight plan to the system, one not initially foreseen in the scenario.

This simple example is already in use and much appreciated by validators of Air Traffic Control systems since it lets them create geographically well placed flights on demand to validate quite precisely the behavior of a software element under target conditions. This is only one illustration of the greater productivity that the use of a dynamic object language, like ILOG TALK, confers on a tool like DyVE.

The validator could also build and send messages, either to move an agent along within its own logic, or to evaluate the behavior of the element to validate when it is confronted with a message unrelated to the system logic.

Commands even offer the validator access to services. The validator can thus, for example, simulate breakdowns by stopping the sending of messages or by corrupting them on the fly. The validator could put in tools to spy on the data being exchanged, or command tracing services. In addition, since the command manager is also the ILOG TALK command line, the validator may use all the debugging tools offered by the ILOG TALK development environment.

6.5 Substitution: how it works

Since the DyVE command line is integrated with EMACS and the development environment, we can actually go further than a simple function call. While a scenario is being played out, the validator can write a short program, for example to insert it at the reception of each message. Obviously, instead of typing it, the validator can load and use a module already written with (why not?) a graphic interface, since ILOG TALK binds to ILOG VIEWS.

Since we use ILOG TALK, a dynamic language, in many parts of the tool layer, even in certain agents and since the *engine* service, with its rule bases, introduces a form of *late binding* in the agents, even in those written in C++, the validator could also redefine functions from the DyVE command line and thus dynamically influence the behavior of services and agents.

Moreover, since the scenario is expressed in TALK, too, these redefinitions could even take place in the scenario, either from loading or deferred to a later time by means of the form `at`.

This possibility of redefining parts of the code to influence the behavior of agents and services is not just a gimmick. On the contrary, it involves *regulation through substitution*, letting the validator get just the results he or she wants while greatly reducing the size of the models making up the simulator.

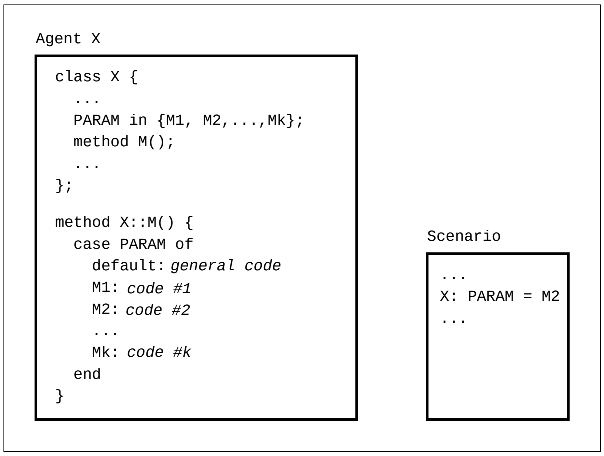

Figure 3: *Controlling an agent through parameters.*

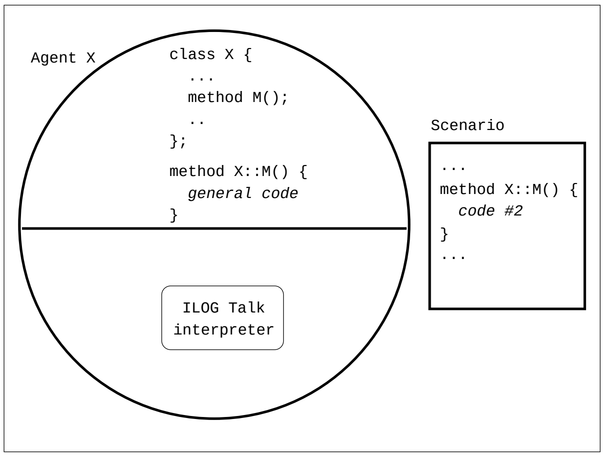

Figure 4: *Controlling an agent through substitution.*

Figures 3 and 4 highlight the differences between regulation through substitution and conventional control through parameters. In the illustration, the agent X is implemented by the class `X`.

Among other things, it has a method `M` that can be regulated by the validator.

With conventional control through parameters, the designer of the model of X must foresee the various regulations that the validator might want to use, then enumerate them all in a parameter, `PARAM`. The body of the method `X::M` will reflect the various possible values of `PARAM`. The scenario will consist of assigning the code of the variation chosen in the parameter; likewise, with the command. This way of doing things begins to be burdensome if the validator ever wants to try a new variation. He or she has to agree precisely with the designer of X, then wait for the implementation; all the while the designer of X is trying to insure coherence and consistency with the rest of the model in all its possible variations.

In contrast, with regulation through substitution, the model of X does not introduce `PARAM`. The body of `X::M` contains only the general case of the method. If the validator wants to use a variation of the method `M`, he or she will substitute the code of this variation for the original code in the scenario or command line. In this way, the code for the model X remains simple, and it is on the basis of each execution that the behavior is redefined by the validator him- or herself expressing precisely what he or she wants to see. Of course, this way of doing things does not preclude help from the designer of the model for X.

6.6 Towards validating integration

The services of the tool layer and the expert layers in the agents are implemented as libraries. The architecture (that is, the kind and number of agents) of a simulator is not implemented by a main program, but rather it is defined in a section of the scenario. This convention makes DyVE independent of the software system that it simulates or validates.

Thus the basic code in DyVE is quite short since it is limited to the *shell* and *session* services. At start-up time, DyVE reads the scenario about the agents and managers that make up each of the processes in the simulator; then it loads the relevant libraries to implement those agents and managers into its own process. The tool layer services used by the agents and managers (and only those services) are loaded as side-effects by means of the dependences established by the ILOG TALK development environment at compile time.

The mechanism of dynamic loading works equally well with libraries that result from TALK code or from C++, or even from a mix of the two, because of development tools put in place by DyVE. Additionally, libraries of utilities can also be loaded during an exercise, for example, such a utility as a window module for capturing messages. It will automatically insure loading the binding for ILOG VIEWS, if needed.

By putting the definition of the architecture of the simulator into the scenario (the place where the validator can express preferences) we leave the choice of agents to use for a given validation session up to the validator. The validator also chooses how many agents, how to distribute them among processes and among available machines. We also give the validator the choice among various existing implementations for each agent, corresponding to various versions, to various degrees of realism or automation, from wholy manual (across various graphic interfaces) up to totally automated (through parameters in the scenario to choose various behaviors).

By exploiting these choices about the architecture to the limit, we can use the simulator to produce integration tests by including prototyped versions or final versions of the same agent in successive simulation sessions in the course of validation.

7 Conclusion

In this article, we have described the DyVE workshop for building environments for validation through simulation. This workshop embodies libraries of services at the tool level along with foundation libraries for Air Traffic Control systems. It also offers a validation methodology, which consists of building a simulator on a general model, then varying the model by substituting code.

To make the validation steps more efficient, DyVE introduces dynamic qualities at multiple levels: at the level of the software system as a whole, by allowing the architecture to be composed from the scenario; at the level of the code, by using code substitution to express variations needed during validation; at the level of model execution, by putting a command line at the disposal of the validator—a command line with the expressive power of the scenario and capable of controlling agents as well as services.

Dynamic qualities have been integrated through our choice of the dynamic object language, ILOG TALK. DyVE takes its interpreter for its command line and for its scenarios from TALK. Since ILOG TALK can absorb C++ libraries, it offers a dynamically usable interface to them so that new users can reuse their C++ code. Dynamic loading of libraries, along with ILOG Talk development tools, make DyVE modular and flexible.

In that way, we have achieved a reactive validation tool integrated with the development environment—a facility that reduces the amount of time invested in validation, all the while making validation more useful.

References

[1] Guido van Rossum and Jelke de Boer. Interactively testing remote servers using the Python programming language. *CWI Quaterly*, 4(4):283–303, December 1991.

[2] Harley E. Davis, Pierre Parquier, and Nitsan Séniak. Sweet Harmony: The Talk/C++ Connection. In *ACM Conference on Lisp and Functional Programming*. ACM SIGPLAN, ACM Press, 1994.

[3] Harley E. Davis, Pierre Parquier, and Nitsan Séniak. Talking about Modules and Delivery. In *ACM Conference on Lisp and Functional Programming*. ACM SIGPLAN, ACM Press, 1994.

[4] ILOG. *ILOG Rules 3.0 Reference Manual*, 1995.

[5] ILOG. *ILOG Talk White Paper*, June 1995.

[6] ILOG. *ILOG Talk 3.2 Reference Manual*, 1996.

[7] ILOG. *ILOG Views 2.2 Reference Manual*, 1996.

[8] Thomson. *DyVE 1.0 Software User Manual*, June 1996.