Proceedings of the 2nd IFIP/ACM Working Conference on Component Deployment (CD), Edinburgh, Scotland, UK, May 2004, pp. 113-128

Dynamic Deployment of Executing and Simulating Software Components

Alexander Egyed

Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
aegyed@ieee.org

Abstract. Physical boundaries have caused software systems to become less monolithic and more distributed. The trend is progressing to a point where software systems will consist of numerous, loosely-coupled, heterogeneous software components. Increased software dynamism will allow these components to be composed, interchanged, upgraded, or even moved without shutting down the system itself. This form of dynamism is already well-supported through new programming constructs and support libraries (i.e., late binding, introspection); however, we are currently ill-equipped to analyze and simulate those kinds of systems. This paper demonstrates that software dynamism requires not only new modeling constructs but also new simulation environments. While in the past, simulation merely mimicked some real-world behavior, we argue that in the future it will become necessary to intertwine the model world with the real world. This will be essential but not limited to cases where (1) one has incomplete access to models (i.e., proprietary COTS components), (2) it is too expensive to model (i.e., Internet as a connector between software components), or (3) one has not complete faith in models (i.e., legacy components). This paper presents our approach to the concurrent execution and simulation of deployed software components. It will also discuss key differences to "traditional" simulation, emulation, and other similar concepts that are being used to integrate the model world with the real world.

1 Dynamism in Today's World

Today, systems are dynamic entities. A modern CPU can reduce its processor power to conserve energy when its energy supply (i.e., battery) is low. A cell phone adapts to different geographical zones (environments) it is taken into. Examples like these show that systems adapt dynamically to their environment or that systems adapt to dynamic environments. There are many reasons why such dynamic behavior is beneficial but the primary reason is that systems continue to operate even if their surroundings vary or are not as expected.

Although it is as much software as it is hardware that gives systems their flexibilities, it is still not often possible to take "pieces of software systems" during run-time and manipulate them. We believe that this will change in the future. Existing tech-

nology, such as late binding between software components, already makes it possible to compose software components into systems dynamically. This technology also makes it possible to replace, upgrade, or move software components dynamically without requiring the overall software system to shut down.

The key benefit of component dynamism is increased flexibility in constructing and maintaining software systems. The benefits range from better reuse of legacy and commercial-off-the-shelf (COTS) components, easier upgrading of older versions of components, simpler replacement of faulty components, added flexibility in distributing components, ability to move components between processing devices (wearable software), and increased generality of component interfaces.

These are not new abilities. Distributed systems, for example, long had many of these flexibilities relying on network protocols. What has changed today is the proliferation of many new component composition technologies, such as remote method invocation [18], COM [21], CORBA [13], or Beans [17] that provide similar flexibilities with different cost-benefit trade-offs.

1.1 Modeling and Simulating Dynamism

Modeling is a form of handling the complexity of software development. To date, we have available a rich set of modeling techniques to cover development aspects such as requirements engineering, software architecting, designing, risk management, testing, and others. The chief benefit of modeling is to support the early evaluation of functional and non-functional properties of software systems. However, short of programming, only the simulation of those models can ensure certain desired qualities and functionalities.

Though we believe that software components will remain complex entities, it is foreseeable that software components in general will become more independent of one another and more configurable to support much needed flexibilities., This poses new challenges to the modeling and simulation of software components and it raises the issue how to represent a system's environment(s).

To increase component flexibility (composability, replaceability, etc.) it is not sufficient to build software components with better user interfaces and export/import features. Instead, the increasing number of uses of a software component have to be taken in consideration while designing it. During execution, for example, components have to be able to reconfigure themselves into different modes of operation (e.g., self healing [20]) corresponding to environmental changes (temporary absence of another component). During design, the designers may spend considerable time and effort in understanding the many environments a component may find itself in. Potentially more effort may be spent in understanding a component's environment than in understanding that component's internals.

This paper emphasizes the modeling and simulation of dynamic software systems. We will briefly discuss some of the new modeling constructs that are required to model dynamic systems but we will concentrate primarily on the execution (simulation) of software models, which is particularly affected by software dynamism.

While both static analysis (e.g., model checking) and dynamic analysis (e.g., execution/simulation) are essential for software modeling, it is simulation we believe to be the most weakly prepared. In a world where a component's environment is becoming more complex, modeling that environment will become more so expensive. Even worse, models may not be available in cases where COTS or legacy software is being used. This paper shows that the lack or incompleteness of models does not prevent the simulation of the overall system. We will present a dynamic simulation environment that can link real software components with simulated software components so that the simulated components interact with real components in absence of their models. Note that we use the terms *component* and *system* interchangeably. A system is a collection of software components but a system can be a component.

2 Intertwining of Model World and Real World

Design-time validation is effective on systems that are modeled formally and completely. If such systems incorporate un-verified components (e.g., COTS software, legacy software, or other third-party software) or operate on un-verified environments (e.g., hardware, network, middleware, operating system) then these unverified elements add potentially unknown constraints [5]. These unknown constraints are uncertainties that limit our ability to reason precisely in their presence.

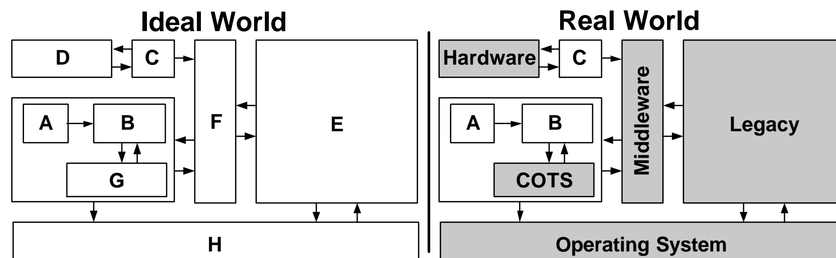

Figure 1. Ideal World where all components are defined (left); Real World with some model descriptions missing (right)

Figure 1 depicts two different perspectives on a seemingly identical, hypothetical system. Both perspectives depict the system's architecture in form of their components and interactions. The left perspective assumes that all components are well-defined (i.e., complete models). Simulating this idealized architecture is simple because of complete knowledge. The right perspective differs only in the use of unverified components. There, some components are well-defined (e.g., component A) but the reuse of existing knowledge introduces undefined components (e.g., COTS). It is typical that model definitions are unavailable for unverified components (e.g., hardware, COTS software, middleware, legacy code, and operating system). This is problematic during reuse because it limits early analysis and simulation. Several choices are available for dealing with this limitation:

1) Disallow the use of unverified components/environments
2) Model unverified components/environments explicitly
3) Ignore the effects of unverified components/environments or make default assumptions about their effects (e.g., ignore the middleware)
4) Prototype, emulate unverified components/environments

The first choice is clearly impractical. It has become highly desirable to reuse existing software into software systems as it can significantly reduce development cost and effort, while maintaining overall software product quality. The second choice is ideal but it is also very costly and time consuming; especially in cases where the environment is more complex than the modeled part of a software system. It is done occasionally in special-purpose domains (e.g., safety critical systems) but it may not be feasible always to create explicit, precise models of unverified components/environments (e.g., due to the proprietary). The third choice is the most practical approach for many situations. Typically, it does not matter (and should not matter) what kind of middleware connects two or more software components. The effects of the middleware are then ignored or trivialized. However ignoring its effects bears risks (e.g., time delays imposed by middleware in real-time systems). Similarly, making default assumptions may trivialize the effects of components/environments (e.g., if the middleware communication is based on a UDP network then there is no guarantee that messages arrive in the same order as transmitted; if a RF network is used then there is not even a guarantee that messages arrive at all).

Simulation requires a sufficiently complete model of the software system and all relevant environmental aspects that affect it. In dynamic systems, the problem is an increasing lack of software component models (e.g., COTS component, legacy) and a lack of environment models (e.g., hardware, operating system). In the very least, this reduces our ability to simulate dynamic systems. In extreme cases, simulation becomes impossible. In Figure 2, we refer to a pure modeling approach as "idealized modeling." The simulation of idealized models is the simulation of software components together with the simulation of their environment (see bottom, right of the table).

	Real Environment	Simulated Environment
Real Component	construction	emulation testing support
Simulated Component	protoyping ???	idealized modeling

Figure 2. Integrating Real and Simulating Components and Environment

If complete models are not available then several choices exist. It is possible to execute real software components (e.g., deployable component implementations) in context of some emulated environment. This is a technique used during testing to "artificially engineer" test scenarios that are normally hard to enact (e.g., simulate test scenario that are too costly or too dangerous to do in reality). This testing support is

often referred to as simulation (see upper, right of Figure 2) but this form of simulation is limited to specialized environmental conditions. Indeed, developers tend to create different specialized models of the same environment to support distinct test scenarios. These specialized environments are typically *not* adequate to represent an environment generically.

Prototyping is another common form of testing dynamic software systems. The middle, left of Figure 2 indicates that prototyping tests real, albeit simplified, software components in context of a real environment (e.g., environment of a deployed component implementation). In principle, prototyping is not very different from implementation although its simplified realization of the software component and its early availability in the software lifecycle gives it the flair of simulation. However, there are key reasons why prototyping is not a substitute for simulation.

1) prototyping language is a programming language
2) abstract modeling concepts are not present in a prototype
3) hard to translate model to programming language
4) harder to re-interpret prototype changes in terms of model changes (consistency problem between prototype and model)
5) hard to observe prototype behavior from a model's point of view
6) prototypes emphasize the interaction among to-be-developed components and their environment ignoring key architectural decisions
7) temptation to keep prototype and throw away model

3 Dynamic Simulation

Many forms of dynamic behavior can be modeled today. Even simulation support exists that can mimic such dynamic behavior. However, dynamic systems exhibit many forms of predicable and unpredictable behavior that are imposed from the "outside" (the environment). The previous section discussed some of these outside influences and concluded that they often cannot be modeled. Ignoring dynamic behavior imposed through the environment may be valid in some cases but bears a risk. That is, in a world where environmental conditions drive component behavior, it becomes increasingly important to understand a component's environment – the ways components can be composted, moved, interchanged, or upgraded. This is especially important for modeling and simulation because a *component's environment may become more complex to model and to simulate than the component itself*.

Unmodeled dynamic behavior imposed by the environment diminishes our ability to simulate dynamic software systems. Not only can it be very expensive to model unavailable components, environments, and infrastructures but once available there is no guarantee of adequacy or correctness. Moreover, the very nature of modeling implies course grain descriptions. Details needed for fully realistic simulation may not be captured in models. This poses the challenge on *how to simulate dynamic systems adequately*? This section introduces dynamic simulation and discusses how the real world is made to interact with the simulated world to substitute models that are unavailable, inadequate, or potentially incorrect.

In dynamic simulation, simulating components may interact with real, executing components. This may serve many purposes such as simulating a component interacting with a real component (e.g., COTS) or simulating a component interacting with another simulating component through a real, intermediary component (e.g., middleware). Dynamic simulation falls into the lower left area of Figure 2.

Being able to intertwine the execution and simulation of real components and simulating components provides significant flexibility to analyzing dynamic software systems. Such flexibility cannot be accomplished without crossing the border between the simulated world and the read world somehow and somewhere. This is a problem because a simulator typically cannot interact in arbitrary ways with the real world and neither can a real component interact with a simulation.

Thus, if a simulating component sends an event to a real component then the simulator must be aware that the recipient of the event is a real component. The simulator must then pass on the event to some mediator that understands both the simulating world and the real world. The mediator will receive events and forward them to the recipients. The mediator will also acts as a recipient of events from the real world to forward them to the simulated world.

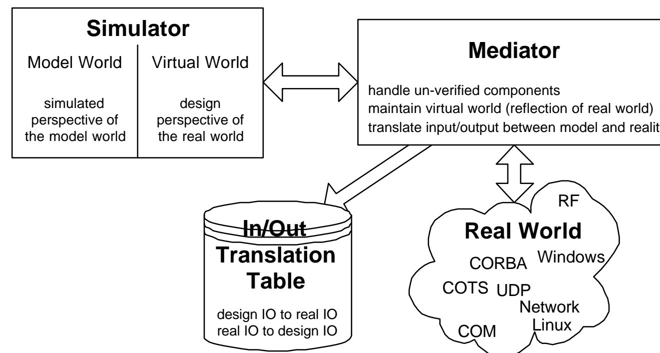

Figure 3. Dynamic Simulation Concept

Figure 3 depicts dynamic simulation schematically. It shows the Mediator as proxy between the Simulator and the Real World (a virtual simulating component) to facilitate interaction between both worlds. The translation table defines how to translate real interactions into simulated interactions and vice versa. More than one simulator may interact with the real world.

While dynamic simulation is simple in principle, there are several key challenges to master. The following introduces an example system to support the subsequent discussion on the key challenges of dynamic simulation in Section 5.

4 Video-On-Demand Case Study

We illustrate the modeling of software dynamism and the dynamic simulation using a video-on-demand software system developed by Dohyung Kim [2]. The video-on-demand system, or VOD system, consists of a movie player, a movie list server, and a commercial data file server. The details of the software system are proprietary but we can discuss some of its modeling aspects.

Figure 4 depicts a course grain, architecture-level overview of the VOD system. The player itself is a client component that is installable and executable on distributed nodes. The movie player consists of a display component for showing the movie (MovieDisplay) and a streamer component for downloading and decoding movie data in real-time (VODPlayer). The movie list server is essentially a database server that provides movie lists and information on where to find those movies. The movie list server handles requests from new players (VODServer) and it instantiates separate handler for every player (ClientHandler). Movies are kept on a file system.

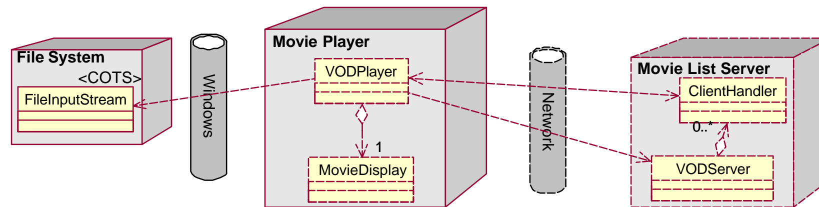

Figure 4. Component Model of Video-On-Demand System (VOD)

The VODPlayer initiates interaction with the VODServer. Upon construction of the ClientHandler, the VODPlayer then interacts with the ClientHandler. The player starts downloading movie data only when a movie is selected. Movie data is downloaded from a file system.

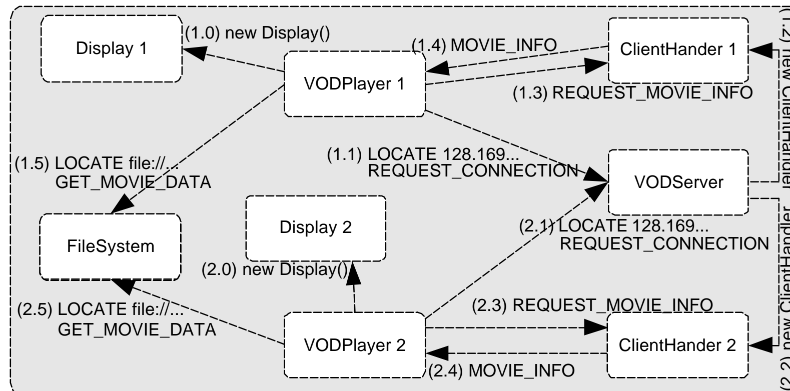

Figure 5. Instance Model During Simulation

The normal operational mode of the VOD system requires the movie list server and the file system to be operational for the player to function. Many players may be executing concurrently on different devices. Figure 5 depicts one possible configuration of two player instances interacting with the movie list server (VODServer) and the file system. The figure shows two instances of the VOD Player, both interacting with the same VOD Server. Two instances of the ClientHandler exist (created by VODServer) to interact with the two instances of the VODPlayer (i.e., one for each player). The role of the client handler is to return details about movies and whole movie lists upon request. Once movie data is available, the player downloads and displays the movie. A possible sequence of events is indicated through the numbers. The example is somewhat simplified from the original for brevity. See also [3] for a more detailed description of the VOD system with architectural models.

5 Challenges of Dynamic Simulation

5.1 Handling Modeling Commands

This work is based on a modeling language called SDSL [3], Statecharts for Dynamic Systems Language, that adapts ADL-like component descriptions (ACME [4], C2SADEL [19], or Darwin [12]) and integrates behavioral semantics similar to Harel's Statecharts (Statecharts [6], Darwin/LTSA [11,12], or Rapide [10]). A rich language was built to express internal component behavior and interactions among components. Unlike Statecharts, SDSL can model advanced dynamic constructs like remote method invocation, late binding, introspection, instance localization. The dynamic instantiation and destruction of components is also supported. It is not necessary here to understand the SDSL in detail. Please refer to [3] for details.

A component maintains references to other components through ports. Ports are a widely used modeling concept (e.g., [14,16]) because a port provides a strong separation between the internals of a component and its environment. For example, the VOD Player in Figure 5 initially contacts the VOD Server to initiate communication. Instead of handling the incoming request, the VOD Server instantiates a helper (client handler) and delegates to it the work of handling future player requests. The player is never aware of this context shift which is elegantly hidden behind ports.

Ports are very useful for dynamic simulation because they also simplify the task of separating real and simulating components. Instead of augmenting SDSL to understand the difference between real components and simulating components, ports make them appear identical. For example, the VOD player communicates with the VOD Server and/or ClientHandler through the port. The player does not know or care whether the data in the port is forwarded to a real component (through a mediator) or a simulated component (without a mediator). Similarly, the VOD player requests data through the port ignorant of where the data really came from.

5.2 When to Use Mediators

A real component is hardwired to interact in a specific way (or set of ways) with other components. Unfortunately, a simulating component in a dynamic system has two choices. If a simulating component interacts with another simulating component then it needs to follow a different interaction strategy than if a simulated component interacts with a real component. In the first case, a mediator is required whereas in the second case not. This problem is analogous of local versus remote calls in container-based systems (e.g., COM, CORBA) with distributed components.

An easy solution to this problem is to create a modeling language that lets the designer decide the interaction strategy. This could be accomplished by, say, creating two different commands for sending an event or by using a with/without mediator flag (i.e., parameter). Unfortunately, this has the disadvantage that the model description of a component is affected by how it is being simulated. This seems unreasonable in that the use of the mediator during simulation should not affect any functional or non-functional property of any component (i.e., no model change).

Perhaps a better way of determining whether to use mediators is through their existence and availability. Flags could be added to mediators to indicate this. If a mediator is available then it will be used; otherwise not. This solution is much better but has one significant drawback. The decision of whether to use mediators is made statically. It is thus not possible to customize the use of mediators for individual instances. For example, if Player 1 (Figure 5) is simulated on a different machine than, say, the Movie List Server and Player 2 is simulated on the same machine then Player 1 requires a mediator to interact with the server while Player 2 does not.

If the use of mediators differs among instances of the same component then simulation needs to make a decision dynamically. We offer five strategies:

1) Simulation (no mediation) first: the component always interacts with another component without a mediator. Only if the simulator fails to interact with the component then a mediator is used (if available).
2) Mediation first: reverse of above
3) Simulation only: do not use mediation even if available
4) Mediation only: do not use simulation even if available
5) Decision hardwired: resolution strategy is hardwired into the mediator in form of an algorithm that is executed during the instantiation of a component

A designer may define the strategy on a model, on individual components, or even on individual ports within components. For example, the whole model can be defined *simulation first* except for component X, which is defined *mediation only*. Strategies 3) and 4) are useful if all instances should be treated equally. Strategy 5) can individualize the behavior of instances.

5.3 Maintaining Dependencies between Real Instances and Simulated Ones

While the previous discussion pointed out how information is sent and received through ports, it did not answer how to maintain correct associations between simu-

lating components and their real interfaces for correct communication. Recall the discussion about mediators and the dual role they play to facilitate the interaction between the simulating world and the real world.

During dynamic simulation, components get instantiated in three different ways. A real component, for which no model exists, is instantiated in the real world only. Similarly, a model component, for which no implementation exists, is instantiated in the simulated (model) world only. However, a model component that interacts with a real component (its environment) requires a model description and an implementation of its real interface. Only it is capable of communicating with both the simulating world and the real world. The others are limited to interacting with components of their respective worlds only.

The model description of a component captures the internal functionality of the simulating component and how it interacts with its ports. If the model component has a real interface then it mediates the component's ports (only those ports that are meant to communicate with real components) and translates data/command contents (provided through the simulation) into real data/commands (and vice versa). The top of Figure 6 shows the result of instantiating the player and the movie list server (normal boxes). Both have real interfaces (boxes with double side bars) attached to ports (circles) underneath them to indicate that they talk to both the real world (the network between them) and the simulated world. The player also instantiates a display component that interacts only with the simulating player in the model world.

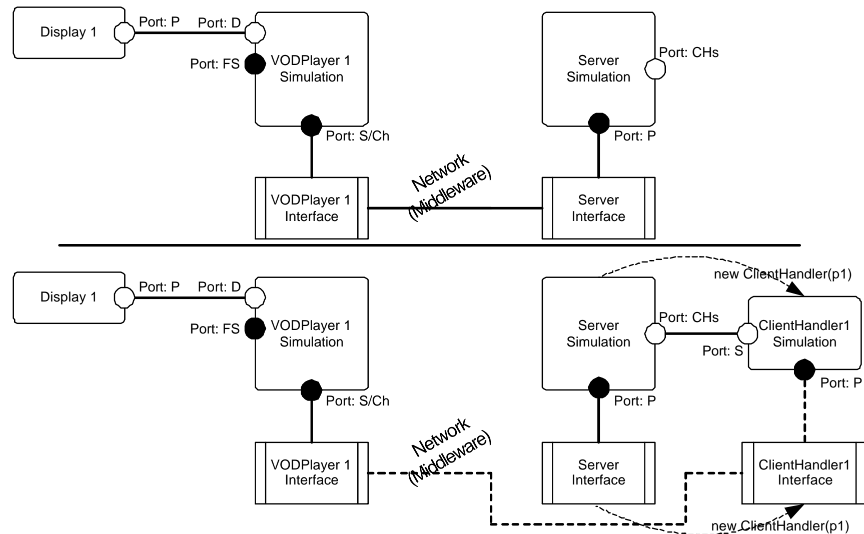

Figure 6. Parallel Construction of Model Components and Implemented Interfaces causes a Dependency Problem

Since there are many simulating (model) components that interact with real components, it is vital to maintain correct associations between simulating components and their interfaces to the real world. Otherwise, the routing of data/commands be-

tween ports and interfaces is erroneous. In Figure 6 (top), this was indicated through lines connecting real interfaces with ports of simulating components. During startup it is typically easy to maintain correct associations because the instantiation of a model component coincides with the creation of its real interface. Once created, simulated data/commands placed into ports are picked up by listeners, translated into real data/commands, and executed by the interface.

This solution works nicely for as long as the simulating components do not instantiate other simulating components that also interact with real components and engage in some form of data exchange that is relevant to the real world. As an example, consider the player/server communication once more (bottom Figure 6). After the player contacts the movie list server, the server creates a client handler to handle this and future requests from the player. ClientHandler, much like Server, is a model component that interacts with the player through the real world. Therefore, a real interface for the client handler must be instantiated together with the instantiation of the simulating client handler. Herein lays the problem. A reference to the port of the player is passed from the server to the client handler. The real interface of the client handler thus requires the real reference (a socket variable) that is defined only in the real interface of the server. How is it possible to pass this reference (the socket) along with the simulated instantiation of the client handler?

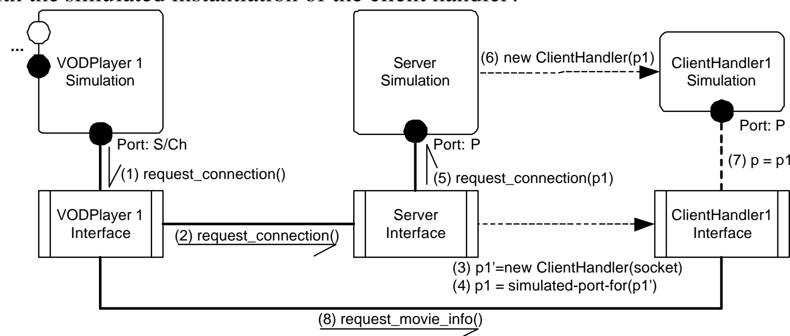

Figure 7. Maintaining the Dependency between Simulation and Reality

Obviously, the real interface of the server must create the real port of the client handler (called ClientHandler1 Interface) in order to pass along the real socket variable. Also, the simulated server must instantiate the simulated client handler to pass along the simulated socket variable. Yet, the mediator must establish an association between these separately created elements. This problem is further complicated in that requests for connection may or may not be routed through mediators (recall Section 5.2) and that the model description of player, server, and client handler should be identical in both cases.

Our solution is to have the interface of the server create the interface for the client handler, create a simulated port for the new interface, and pass the port as part of the simulated instantiation of the client handler. Figure 7 depicts the order of events for this solution. When the server interface receives a request_connection from a player, it creates a real port as well as a simulated port. The simulated server is then notified

of the request connection event with the parameter being the simulated port of the player (i.e., the simulated port is a references to the real port). The server then instantiates the client handler, passes along the simulated port, and substitutes its port to the player with the one passed along. This solution is consistent with the implementation where the client handler receives the real port as part of its construction.

6 Discussion and Validation

We applied our simulation language SDSL and its dynamic simulator SDS on four major case studies to date. Three case studies involved third party components and one was developed in-house. All our case studies were based on real software systems, which were used to evaluate how well the simulation mimiced them. Our case studies used a wide range of dynamism technologies such as COM, late binding, remote method invocations, and networks. The largest SDSL model we created to date is the VOD system with about 40 states and 50 transitions.

The cost of building SDSL models is proportional to the required level of detail. SDSL models can be very generic but also very specific. The effort of building the VOD system was about 7 person hours. The effort of building the mediators and translators was about 14 person hours. The latter is surprisingly little given the size and the complexity of the VOD system. We attribute this to extensive code reuse. For example, we were able to reuse over 70% of the server interface code from the real server and 64% of the player interface code from the real player. In total, less than 130 lines of code had to be added or modified.

Depending on the complexity of a component, simulating it within a real-world environment can be significantly cheaper than building it. The real player has about 4000 lines of code, its simulation less than 300. Naturally, the simulation also requires the model. If the real system would not have existed then 70% of the interface code could have been reused during code generation.

Initially, we constructed a model of all VOD components. Once we had the model of the VOD system, we built the component interfaces required to enable dynamic simulation. Through dynamic simulation we discovered major inconsistencies between the model and the system because real components and simulated components did not interoperate. It took only little effort to detect, locate, and fix those flaws to a point where we have strong confidence in the model's correctness.

We also used our simulator as a test environment for real software components. For instance, we had the simulator instantiate a large number of simulated players (almost) instantaneously to see how the real servers handle the load. Or, we had the simulated server send bogus answers or scrambled text to see how the players react. This form of stress testing required no additional coding but would have been much more costly if we had done it purely in the real world. Moreover, we could define those hypothetical test scenarios in form of SDSL models and we could test them directly on real components. We find that this is a very efficient way of testing model scenarios.

We found that SDSL and its simulator make models more active participants during design (simulation), coding (dynamic simulation and code generation), and testing (scenario testing). During design, simulation helps finding flaws quickly. During coding, dynamic simulation and code generation enable rapid application development. And during testing, model test scenarios can be validated directly on real components. As a result, we intertwined programming and modeling to a point where it is hard to distinguish them. In fact, in another case study we adopted a simulated COM component as a real component because it satisfied all our requirements [20]. We saw no need of implementing that component. It must be noted that these extended features of dynamic simulation overlap with prototyping, emulation, and testing.

Despite all advantages, dynamic simulation also has downsides. Traditional simulation mimics the real-world in a reproducible way. Thus, re-running a specific simulation scenario produces the exact same results every time. Unfortunately, dynamic simulation opens simulation to the unpredictability of the real world. With a real network between two simulated components no two simulations will be exactly alike. Simulating scenarios during dynamic simulation is thus comparable to testing during coding. Although we found that dynamic simulation is often good enough, we do not believe that it can replace closed-world simulation or static analysis.

7 Related Work

There are only few behavioral modeling languages available that can handle dynamism in some form. Harel-Gery [7] combined class diagrams with statechart diagrams to enable design time dynamism and their tool, Rhapsody [8], incorporates design-time dynamism constructs. This approach does not integrate the model world and the real world during simulation.

Rhapsody's integration with object models makes it a suitable candidate to model dynamism in the context of UML (i.e., construction, destruction). However, modeling constructs, such as ports, are not supported. This limits the dynamic behavior of Rhapsody to direct method invocations (i.e., procedure calls). Even in cases where architectural languages (ADLs) and their advanced concepts have been mapped successfully to UML (e.g., C2 [19] to object model mapping) [1] this mapping also changed the meaning of those objects (that is a main reason why stereotypes were used). For instance, in C2 one component is not aware of components next to it and thus cannot refer to it directly by name. An object model representing a C2 component model thus cannot make use of Rhapsody's statechart simulation capabilities.

Two ADLs that have stressed the ability to describe dynamism require some mention. First, the event-based model of Rapide [10] has been used to describe architectural components and the events they are exchanging. Its tool suite can then be used to analyze event patterns to identify potential problems (e.g., an invalid causality relationship). Again, although we are unaware of any other efforts to provide runtime (model) dynamism, Rapide only supports some forms of design-time dynamism like the creation of components dynamically. The use of Rapide for dynamic model-

ing purposes is additionally hampered by its tight links to the rest of Rapide; this is much the same criticism as we have for Rhapsody as well.

A second ADL used to describe dynamic effects is Darwin [12]. The language is certainly of a kindred spirit in that it specifies what services are provided and what services are needed for each component. The language is unique for proscribing structural dynamism, by emphasis on lazy binding of (potentially unbounded) recursive structures and, as with both Rhapsody and Rapide, direct dynamic instantiation. Darwin is not event-based, and is incapable of modeling change to as fine a grain size as statecharts.

The ability of SDSL to integrate real-world components and simulated components allows for rapid application development to some degree. Nonetheless, we do not see our approach as another form of prototyping. Prototyping is characterized by quick and dirty programming without adequate design. SDSL models, unlike prototypes are not intended to be thrown away. They can be analyzed statically and dynamically in closed environments. They can also be used for code generation and testing. SDSL models thus have extensive use outside the realm of prototyping.

Dynamic simulation also has some overlap with code generators. Code generators transform model descriptions into code. Models and code can then be executed and analyzed separately. Dynamic simulation also separates models from code through mediators but it does not necessarily distinguish the analysis of code and model as separate activities. Our approach does not have to handle problematic versioning issues during code generation (i.e., overwriting changes) however, it causes its own set of problems that were discussed previously.

8 Conclusion

Simulators play a vital role in validating component dynamism. They enable the rapid testing of dynamic software systems with minimal coding. Simulation allows the safe exploration of a proposed solution in an environment that shields from physical harm and monetary harm [15]. We know from previous studies that simulation is a very cost effective and economical way of building and validating software systems [9].

This work proposed dynamic simulation as a complement to validation and testing under situations where it is uneconomical or infeasible to model un-verified components. Dynamic simulation combines the simulation of modeled components with the execution of deployed, un-verified components within un-verified environments. It only requires the modeling of newly developed components and uses existing components and infrastructure. It is complementary to other forms of static and dynamic analyses and can be used for prototyping or testing. Careful attention was given on how to separate models from code during dynamic simulation. We use mediators and translators to ensure that neither model nor code needed to be tailored towards dynamic simulation scenarios.

Still, the concept of dynamic simulation is not revolutionary. It borrows heavily from prototyping, emulation, and even testing. Nonetheless, this paper contributed

strategies on how to handle problems associated with the dynamic behavior of software components and on how to maintain dependencies between the model world and the real world correctly. To the best of our knowledge, these issues have not been explored in the past for these domains.

Dynamic simulation also has disadvantages. Its main downside is that it opens modeling to some of the unpredictability of the real world. This is not always problematic but it limits the usefulness of dynamic simulation in some cases. Future work is needed to further validate our approach to determine the trade-offs between closed and dynamic simulation and the cost of building mediators. It is also intended to investigate whether component simulation can be done concurrently with their real counterparts to reason about state and consistency. Furthermore, it is future work to generalize the current tool support for dynamic simulation. To date, mediators have to be created manually. We believe that this could be automated partially.

References

1. Abi-Antoun, M. and Medvidovic, N.: "Enabling the Refinement of a Software Architecture into a Design," *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*, October 1999.
2. Dohyung, K.: "Java MPEG Player," *http://peace.snu.ac.kr/dhkim/java/MPEG/*, 1999.
3. Egyed, A. and Wile, D.: "Statechart Simulator for Modeling Architectural Dynamics," *Proceedings of the 2nd Working International Conference on Software Architecture (WICSA)*, August 2001, pp.87-96.
4. Garlan, D., Monroe, R., and Wile, D.: "ACME: An Architecture Description Interchange Language," *Proceedings of CASCON'97*, November 1997.
5. Grundy, J. C. and Ding, G.: "Automatic Validation of Deployed J2EE Components Using Aspects," *Proceedings of the 17th International Conference on Automated Software Engineering (ASE)*, Edinburgh, Scottland, UK, September 2002.
6. Harel D.: Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 1987.
7. Harel, D. and Gery, E.: "Executable Object Modeling with Statecharts," *Proceedings of the 18th International Conference on Software Engineering*, March 1996, pp.246-257.
8. iLogix: Rhapsody at http://www.ilogix.com/.
9. Jackson, D. and Rinard, M.: "Software Analysis: A Roadmap," *Proceedings of the 20th International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000, pp.133-145.
10. Luckham D. C. and J. Vera J.: An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 1995.
11. Magee, J.: "Behavioral Analysis of Software Architecture using LTSA," *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
12. Magee, J. and Kramer, J.: "Dynamic Structure in Software Architectures," *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Oc-

tober 1996.

13. Object Management Group: The Common Object Request Broker: Architecture and Specification. 1995.
14. Perry D. E. and Wolf A. L.: Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 1992.
15. Sanders, P.: "Study on the Effectiveness of Modeling and Simulation in the Weapon System Acqusition Process," *Report, Test Systems Engineering and Evaluation, Office of the Under Secretary of Defense (Acqusition & Technology)*, 1996.
16. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.
17. Sun Microsystems: Java Beans Specification at http://java.sun.com/beans/docs/beans.101.pdf.
18. Sun Microsystems: Java Remote Method Invocation - Distributed Computing for Java. 2001.(UnPub)
19. Taylor R. N., Medvidovic N., Anderson K. N., Whitehead E. J. Jr., Robbins J. E., Nies K. A., Oreizy P., and Dubrow D. L.: A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* 22(6), 1996, 390-406.
20. Wile, D. and Egyed, A.: "An Architectural Style for Self Healing Systems," *under submission to WICSA 2004*.
21. Williams S. and Kindel C.: The Component Object Model: A Technical Overview. *Dr. Dobb's Journal*, 1994.