A Case Against Continuous Simulation for Software Architecture Evaluation

Frans Mårtensson[*], Per Jönsson[*], PerOlof Bengtsson[**], Håkan Grahn[*], and Michael Mattsson[*]

[*]Department of Software Engineering and Computer Science
Blekinge Institute of Technology
P.O. Box 520, SE-372 25 Ronneby, Sweden
{Frans.Martensson, Per.Jonsson, Hakan.Grahn, Michael.Mattsson}@bth.se

[**]Ericsson AB
P.O. Box 518
SE-371 23 Karlskrona, Sweden
PerOlof.Bengtsson@epk.ericsson.se

ABSTRACT

A software architecture is one of the first steps towards a software system. The design of the architecture is important in order to create a good foundation for the system. The design process is performed by evaluating architecture alternatives against each other. A desirable property of a good evaluation method is high efficiency at low cost.

In this paper, we investigate the use of continuous simulation as a tool for software architecture performance evaluation. We create a model of the software architecture of an existing software system using a tool for continuous simulation, and then simulate the model. Based on the case study, we conclude that continuous simulation is not feasible for software architecture performance evaluation, e.g., we identified the need of discrete functionality to correctly simulate the system, and that it is very time consuming to develop a model for performance evaluation purposes. However, the modeling process is valuable for increasing knowledge and understanding about an architecture.

KEYWORDS: continuous simulation, AGV systems, modeling, software architecture, performance evaluation

1. Introduction

The software architecture is fundamental for a software system [6, 10, 18], as it often restricts the overall performance of the final system. Before committing to a particular software architecture, it is important to make sure that it handles all the requirements that are put upon it, and that it does this reasonably well. The consequences of committing to a badly designed architecture could be disastrous for a project and could easily make it much more expensive than originally planned. Bad architecture design decisions can result in a system with undesired characteristics such as low performance, low maintainability, low scalability etc.

When designing the architecture for a system, the architect often has the possibility to choose among a number of different solutions to a given problem. Depending on which solution is chosen, the architecture evolves in different ways. To be able to make a proper decision, the architect needs to identify quantifiable advantages and disadvantages for each one. This can be done by using, e.g., prototyping or scenario-based evaluation [6]. A desirable property of a good evaluation method is high efficiency at low cost.

In this paper we evaluate the use of continuous simulation for system architecture performance evaluation purposes. The main idea is that tools for continuous simulation can be used to quickly create models of different architecture alternatives. These models can then be used, through simulation, to evaluate and compare different architectures to each other. The work presented has been conducted in co-operation with Danaher Motion Särö (referred to as "DMS"). We have co-operated with DMS to use the software architecture of their Automated Guided Vehicle (AGV) system (hereafter referred to as the "DMS system") as a case for the research.

Unfortunately, we found that continuous simulation does not work very well for software architecture performance evaluation. First, when a continuous simulation model is used only average flow values can be used to parameterize the model. This makes the model less dynamic and may have the consequence that the simulation model can be replaced with a static mathematical model. Second, it is impossible to address unique entities when using continuous simulation. This is not always necessary when simulating flows of information, but if the flows depend on factors that are discrete in their nature, for example vehicles in an AGV system, then continuous simulation is a bad choice.

We do however believe that an architecture modeling tool that incorporates some simulation functionality could be helpful when designing and evaluating software architectures. It could, e.g., provide functions for studying data flow rates between entities in an architecture. Such a tool would preferably be based on combined simulation techniques, because of the need to model discrete events.

The rest of the paper is structured as follows: We begin with an introduction to software architectures in Section 2. In Section 3, we discuss some simulation approaches, and in Section 4, we describe two software tools for continuous simulation. Next, Section 5 introduces the AGV system domain. Section 6 describes our attempts to model and simulate the architecture of the DMS system. In Section 7, we have a discussion of our results, and finally, in Section 8, we present our conclusions.

2. Software architecture

There are many different definitions of what a software architecture is, and a typical definition is as follows [10]:

A critical issue in the design and construction of any complex software system is its architecture: that is, its gross organization as a collection of interacting components.

In other words, through the creation of a software architecture we define which parts a system is made up of and how these parts are related to each other. The software architecture of a system is created early in the design phase, since it is the foundation for the entire system.

The components in an architecture represent the main computational elements and data storage elements. The architecture is created on a high level of abstraction which makes it possible to represent an entire subsystem with a single component. The communication between the components can be abstracted so that only the flow of information is considered, rather than technical details such as communication protocol etc. Individual classes and function calls are normally not modelled in the architecture.

With the creation of the software architecture, designers get a complete view of the system and its subsystems. This is achieved by looking at the system on a high level of abstraction. The abstracted view of the system makes it intellectually tractable by the people working on it and it gives them something to reason around [2].

The architecture helps to expose the top level design decisions and at the same time it hides all the details of the system that could otherwise be a distraction to the designers. It allows the designers to make the division of functionality between the different design elements in the architecture and it also allows them to make evaluations of how well the system is going to fulfill the requirements that are put upon it. The requirements on system can be either functional or non-functional. The functional requirements specify what function the system shall have. The non-functional requirements include, e.g., performance, maintainability, flexibility, and reusability.

Software architectures are often described in an ad hoc way that varies from developer to developer. The most common approach is to draw elements as boxes and connections simply as connecting lines. A more formal way of defining architectures is to use an architecture description language (ADL) that is used to describe the entities and how they connect to each other. Examples of existing ADL:s are ACME [9] and RAPIDE [12], which are still mainly used for research. ADL:s have been successfully used to describe new architectures and to document and evaluate existing architectures [8].

A software architecture is, among other things, created to make sure that the system will be able to fulfill the requirements that are put upon it. The architecture usually focuses more on non-functional requirements than on functional ones. The non-functional requirements are for example those that dictate how many users a system should be able to handle and which response times the system should have. These requirements does not impact which functionality the system should provide or how this functionality should be designed. They do however affect how the system should be constructed.

It is important to evaluate different software architecture alternatives and architectural styles against each other in order to find the most appropriate ones for the system. There exists a number of evaluation methods, e.g., mathematical model-, experience-, and scenario-based methods [6]. The process of evaluating the software architectures is mainly based on reasoning around the architecture and the chosen scenarios. How successful this evaluation is depends currently heavily on the level of experience of the people performing it. More experienced people are more likely to identify problems and come up with solutions.

It is during this evaluation phase that we believe that it would be useful to use continuous simulation for evaluating the performance of software architectures. It could be used as a way of quickly conducting objective comparisons and evaluations of different architectures or scenarios. The advantage over other evaluation methods, for example experience-based evaluation, is that simulation gives objective feedback on the architecture performance.

3. Model and simulation

A *model* is a representation of an actual system [5], or a "selective abstraction" [14], which implies that a model does not represent the system being modeled in its whole. A similar definition is that a model should be similar to but simpler than the system it models, yet capture the prominent features of the system [13]. To establish the correctness of a model, there is a need for model *validation* and *verification*. Validation is the task of making sure that the right model has been built [3, 13]. Model verification is about building the model right [3, 15].

A *simulation* is an imitation of the operation of a real-world process or system over time [5, 13, 20]. A simulation can be reset and rerun, possibly with different input parameters, which makes it easy to experiment with a simulation. Another important property of simulation is that time may be accelerated, which makes simulation experiments very efficient [17].

The *state* of a simulation is a collection of variables that contain all the information necessary to describe the system at any point in time [4]. The input parameters to a simulation are said to be the initial state of the simulation. The state is important for pausing, saving, and restoring an ongoing simulation, or for taking a snapshot of it. A simulation model must balance the level of detail and number of state variables carefully in order to be useful. The goal is to find a tradeoff between simplicity and realism [13].

Continuous simulation is a model in which the system changes continuously over time [20]. A continuous simulation model is characterized by its state variables, which typically can be described as functions of time. The model is defined by equations for a set of state variables [5], e.g, *dy/dt = f(x, t)*. This simulation technique allows for smooth system simulation, since time is advanced continuously, i.e. changes occur over some period of time.

When using the *discrete simulation technique*, time is advanced in steps based on the occurrence of discrete events, i.e., the system state changes instantaneously in response to discrete events [13, 20]. The times when these events occur are referred to as event times [4, 16]. In an event-driven discrete simulation, events are popped from a sorted stack. The effect of the topmost event on the system state is calculated, and time is advanced to the execution time of the event. Dependent events are scheduled and placed in the stack, and a new event is popped from the top of the stack [11]. With the discrete simulation technique, the ability to capture changes over time is lost. Instead, it offers a simplicity that allows for simulation of systems too complex to simulate using continuous simulation [20].

Combined continuous-discrete simulation is a mix of the continuous and discrete simulation techniques. The distinguishing feature of combined simulation models is the existence of continuous state variables that interact in complex or unpredictable ways with discrete events. There are mainly three fundamental types of such interactions [1, 11]: (i) a discrete event causes a change in the value of a continuous variable; (ii) a discrete event causes a change in the relation governing the evolution of a continuous variable; and (iii) a continuous variable causes a discrete event to occur by achieving a threshold value.

A software system can be modeled and simulated using either discrete or continuous simulation techniques. When looking at the software architecture of a system, communication between the components can be viewed as flows of information, disregarding discrete events. By leaving out the discrete aspects of the system, continuous simulation can be used to study information flows. It is our assumption that it is simpler to model a software architecture for continuous than for discrete simulation, because low-level details can be ignored.

A good example of a low-level detail is a function call, which is discrete since it happens at one point in time. By looking at the number of function calls during some amount of time, and the amount of data sent for each function call, the data transferred between the caller and the callee can be seen as an information flow with a certain flow rate. Some reasons that make this advantageous are:

- It is valid to consider an average call frequency and to disregard variations in call interval etc.
- Multiple function calls between two components can be regarded as one single information flow.
- Accumulated amounts and average values are often interesting from a measurement perspective.

4. Software tools

Once a model has been constructed, we want to run it to see what results it produces. If it is a simple model then it might be possible to simulate it using pen and paper or perhaps a spreadsheet. But if the model is too complex for "manual" execution then it becomes necessary to use some kind of computer aid. Since we in this paper focus on the possibilities of using continuous simulation in architecture evaluation, we look at GUI-based general-purpose simulation tools that require little knowledge about the underlying mathematical theories.

The first tool that we evaluated was the STELLA 7.0.2 Research simulation tool, which is a modeling and simulation software for continuous simulation that is created and marketed by High Performance Systems Inc. The second tool was Powersim Studio Express 2001, created by Powersim. This is a similar tool that offers more functionality than STELLA as it is based on combined simulation, and has some more advanced features. Both tools are based on the concept of *Systems Thinking* [14] for the creation of models, and both programs are capable of performing the simulation directly in the program and also to perform some basic analysis.

In both STELLA and Powersim, models are constructed from a number of basic building blocks which are combined in order to build a model. The model is simulated by the use of entities that are sent through the model. The flow of entities can then be measured and analyzed. We use snapshots of the STELLA tool, but they look very similar in Powersim and they work in a similar fashion. In Figure 1 we show the five basic building blocks Stock, Flow, Converter, Connector, and Decision Process.

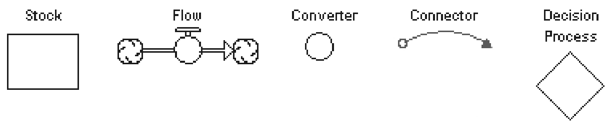


Figure 1: Examples of basic building blocks.

Stocks are used to represent accumulation of entities various ways. *Flows* are used to connect stocks and to enable and control the flow of entities between them. *Converters* are often used in order to modify the rate of flows, and to introduce constants in a model. *Connectors* are used to connect, e.g., stocks and flows so they can exchange information. To make a model less complex it is possible to hide parts of it by using *decision processes*.

Once the model is completed it is possible to run it. Both Powersim and STELLA are capable of accelerating the simulation time. The tools can visualize simulation outputs and results as the simulation runs, e.g., time-graphs, time-tables and value labels. Powersim also has the possibility to export results to a standard file format.

We used Powersim for the following three reasons: (i) Powersim has the ability to check the consistency of the model via the use of units on every flow; (ii) Powersim offers the possibility to create discrete flows and visually distinguish them in a model; and (iii) STELLA crashed repeatedly when we tried to use the decision process functions, and also sometimes even during normal work. The unreliability of the tool made us hesitant to use STELLA.

5. AGV systems

An AGV (Automated Guided Vehicle) system is an automatic system that usually is used for materials handling in manufacturing environments, e.g., car factories and metal works. They are however not restricted to these environments and can be used in very different environments such as hospitals and amusement parks.

An AGV is usually a driverless battery-powered truck or cart that follows a predefined path [7]. A path is divided into a number of *segments* of different lengths and curvatures. There can be only one vehicle on a segment at any given time. The amount of computational power in a vehicle may vary depending on how advanced the behavior of the vehicle is. With more computational power, it is possible to let the vehicle be autonomous. However, computational power costs money, and with many vehicles, a computationally strong solution can be expensive.

The management and control of the AGV system is usually handled by a central computer that keeps track of all the vehicles and their orders. This computer maintains a database of the layout of the paths that the vehicles can use to get to their destinations [7]. With this information it acts as a planner and controller for all the vehicles in the system, routing traffic and resolving deadlocks. The central server gets orders from, e.g., production machines that are integrated with the AGV system.

In order for the AGV system to work it must be possible to find the position of the vehicles with good precision. This is achieved by the use of one or more positioning and guidance systems, e.g., electrical track, optical guidance, and magnetic spots. With electrical track guidance, the vehicle path is defined by installing a guidance wire into the floor of the premises. Optical guidance is achieved for example by the use of a laser positioning system which uses reflectors placed on the walls of the premises in order to calculate an accurate position of the AGV as it moves. Magnetic guidance works by the use of magnetic spots, which are placed on the track. The vehicles have magnetic sensors that react on the presence of the spots.

In an AGV system it is desirable to minimize the communication between the server and the clients. The choice of communication strategy affects the amount of information that is communicated in the system.

An early communication strategy was to let the vehicles communicate with the server only at certain designated places. As a result, the vehicles can only be redirected at certain points, since the server has no control of a vehicle between communication spots. A more advanced way of communicating is via the use of radio modems. The early modems however had very low bandwidth, which imposed limitations on the amount of information that could be transferred. This limitation has diminished as advancements made in radio communication technology have increased the amount of available bandwidth. The next step in communication is to make use of cheaper off-the-shelf hardware such as wireless LAN, e.g., IEEE 802.11b. An advantage with using such a strategy is that an existing infrastructure can be used.

We mention here two alternative ways to design an AGV system, and they are interesting because they represent the extremes of designing a system architecture. The goal of a *centralized approach* is to put as much logic in the server as possible. Since the vehicles cannot be totally free of logic (they have to have driving logic at least), the centralized approach is in practise distributed. However, we may choose different degrees of centralization by transferring modules from the vehicle logic to the server logic. In an entirely *distributed approach* there is no centralized server, thus making the system less vulnerable to failure. This requires all information in the system to be shared among, and available to, all vehicles, which can be realized, e.g., by using a distributed database solution.

6. The experiment

The architecture studied is a client-server architecture for a system that controls AGVs. The server is responsible for such tasks as order management, carrier management, and traffic management. It creates "flight plans" and directs vehicles to load stations. The vehicles are "dumb" in the sense that they contain no logic for planning their own driving. They fully rely on the server system. A more in-depth explanation of the system can be found in [19].

The communication between server and clients is handled by a wireless network with limited capacity, set by the radio modems involved. Topics of interest are for example:

- Has the network capacity to handle communication in highly stressed situations with many vehicles?
- Can the system architecture be altered so less traffic is generated?
- Can the system share an already present in-use wireless LAN?

With this in mind, we decided to simulate the architecture with respect to the amount of generated network traffic. The intention is to provide a means for measuring how communication-intense a certain architecture is.

6.1. The system behavior

The purpose of the studied system is to control a number of AGVs. The AGVs must follow a pre-defined track which consists of segments. A fundamental property of a segment is that it can only be "allocated" to one AGV at a time. Sometimes, several segments can be allocated an AGV to prevent collisions. The primary controlling unit for the system is an *order*. An order usually contains a loading station and an unloading station. Once an order has been created, the server tries to assign a vehicle to the order and instructs the vehicle to carry it out. During the execution of an order, the vehicle is continuously fed segments to drive.

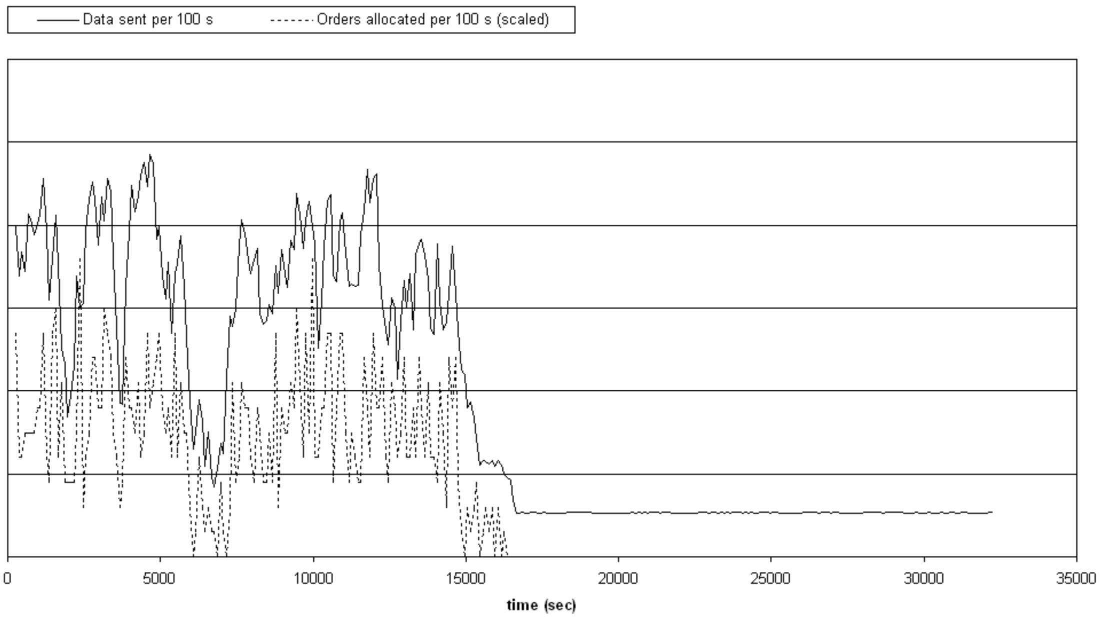


Figure 2: Example of network traffic during a system run, with order allocations superimposed on the traffic diagram.

In certain situations, deadlock conflicts can arise. A deadlock occurs, e.g., when two vehicles are about to drive on the same segment. A traffic manager tries to resolve the deadlock, according to a set of deadlock avoidance rules. As the number of vehicles involved in the deadlock increases, it becomes harder and harder for the traffic manager to resolve the situation.

Each vehicle, i.e., each client, contains components for parsing drive segments fed from the server, controlling engines and steering, locating itself on the map etc. The vehicle is highly dependent on the drive commands sent from the server; if the segment-to-drive list is empty, it will stop at the end of the current segment. If the vehicle gets lost and can't rediscover its location, it will also stop.

The communication between server and clients is message-based. The server sends vehicle command messages to control the vehicles, and the vehicles respond to these with vehicle command status messages. There are also status messages, which are used to report vehicle status.

6.2. The model

We will look at the flow of information over the network in the system architecture. Thus, the communication network plays a central role in the model, and the purpose of all other entities is to generate input traffic to the network. The network traffic in a client-server system has two components; server generated traffic and client generated traffic. However, when measuring the network utilization, the sum is interesting. In our model, the network traffic is modelled as a whole. Further, the network is more or less a "black hole", since the output is discarded.

Prior to constructing the model we had basic knowledge of the behavior of both the server architecture and the client architecture, e.g., which components that communicate over the network. However, we had only vague understanding of what caused communication peaks and which communication that could be considered "background noise". Therefore, we studied a version of the current client-server system. The server system can handle both real and simulated AGVs, which allowed us to run a simulation of the real system in action, but with simulated vehicles instead of real ones (30 vehicles were simulated).

An example of logged network traffic can be seen as the solid line in Figure 2 (the y-axis has no unit, because the purpose is only to show the shape of the traffic curve). In the left part of the diagram, all AGVs are running normally, but in the right part they are all standing still in deadlock. Except for the apparent downswing in network traffic during deadlock, no obvious visual pattern can be found. When analyzing the traffic, we found that normal status messages are responsible for roughly 90% of the traffic, and that the number of status messages and the number of vehicle command messages fluctuate over time. However, the number of vehicle command status messages seems to be rather stable regardless of system state (e.g. normal operation vs. deadlock).

We sought reasons for the traffic fluctuations, and examined the log files generated. We found a connection between order allocations and network traffic. An order allocation takes place when a vehicle is assigned to a new order, and this causes an increase in traffic. In Figure 2, a correlation between the order allocations (dotted line) and the network traffic (solid line) is observed. In particular, during the deadlock there are no order allocations at all. Mathematically, the correlation is only *0.6*, which is not very strong but enough for us to let order allocations play the largest role in the model. The reason for an upswing in traffic when an order allocation takes place is simple; it changes the state of a vehicle from "available" to "moving", and in the latter state the traffic per vehicle is higher.

The network is modelled as a buffer with limited storage capacity. It holds its contents for one second before it is released. Immediately before the network buffer is a transmission buffer to hold the data that cannot enter the network. If the network capacity is set too low, this buffer will be filled. In a real system, each transmitting component would have a buffer of its own, but in the model the buffer acts as transmission buffer for all components.

To model network congestion, the network buffer outlet is described by a function that depends on the current network utilization, e.g., it releases all network data up to a certain utilization limit, and thereafter gradually releases less data as the utilization increases. The data that remains in the network buffer represents data that in a real system would be re-sent. A visual representation of the modeled network is seen to the left in Figure 3. The entity "Network indata" in Figure 3 is at every time the sum of all traffic generated in the model at that time.

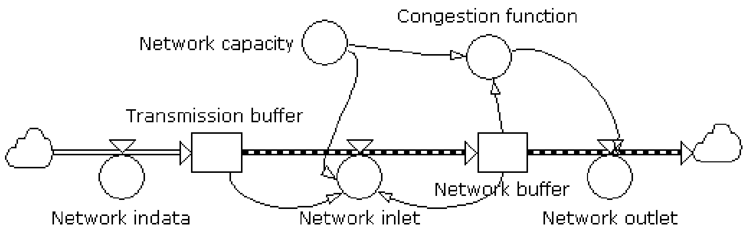
The primary controlling unit for the system is an order, and order allocations generate network traffic. The amount of traffic generated also depends on how many vehicles that are available, processing orders, and in deadlock. Therefore, we need constructs for the following:

- Order generation
- Order allocation
- Available vs. non-available vehicles
- Deadlock

Orders can be put into the system automatically or manually by an operator. We have chosen to let orders be generated randomly over time, but with a certain frequency. Each time an order is generated, it is put in an order buffer. As orders are allocated to vehicles, the number of orders in the buffer decreases. The order component and the order allocator is shown to the right in Figure 3.

For an order allocation to take place, there must be at least one available order, and at least one available vehicle. Then, the first order in queue is consumed and the first available vehicle is moved to the busy-queue. The busy queue contains several buffers to delay the vehicles' way back to the buffer for available vehicles and a mechanism for placing vehicles in deadlock. In the deadlock mechanism each vehicle runs the risk of being put in a deadlock buffer. The risk of deadlock increases as more and more vehicles are put into the deadlock buffer. Once in deadlock, each vehicle runs the chance of being let out of the deadlock again. The chance for this to happen is inversely proportional to the number of vehicles in deadlock. Figure 4 shows the construct describing the vehicle queues and the deadlock mechanism.

The remaining parts of the model fill the purpose of "gluing" it together. They are simple constructs that, given the current state of vehicles, generate the proper amounts of traffic to the network.



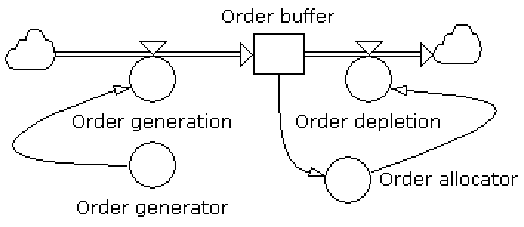


Figure 3: Network component (left), and Order component and order allocator (right).

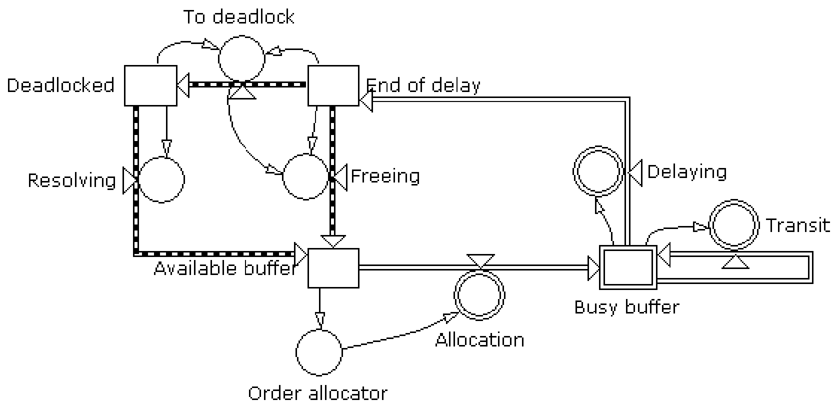


Figure 4: Vehicle queues and deadlock mechanism.

6.3. Simulation parameters and results

In the current system, each vehicle is equipped with a modem capable of transmitting 19 200 bps, while both the network and the server system have higher capacity. We therefore chose to set the network speed to 2 400 byte/s (19 200 bps) in the simulation, since the first step was to build a model that approximates the real system, rather than to study the impact of different network speeds.

In our simulation, we let the order creation probability be high enough to ensure that there is always at least one order in queue when a vehicle becomes available. The average order processing time is set to *230 seconds*. This is based on the average order processing time in the real system when run with 1 vehicle.

The probability for a vehicle to enter deadlock is set to $P_{enter} = 1 - 0{,}99^{x+1}$ where x is the number of vehicles currently in deadlock, i.e., the probability increases as more vehicles enter deadlock. The probability for a vehicle to leave deadlock is set to $P_{leave} = 0{,}2^{y}$ where y is the number of vehicles currently in deadlock, i.e., the more vehicles involved in a deadlock, the harder it is to resolve.

Table 1 contains data points measured in the real system in standby state, i.e., when no vehicles were moving. As seen in Figure 5, the traffic is linearly related to the number of vehicles. The situation when all vehicles are standing still is assumed to be similar to a deadlock situation, at least traffic-wise.

Table 1: Traffic generated in standby state (avg. bytes/s)

No. of vehicles	Status msg.	Command msg.	Command status msg.
0	0	0	0
1	1.2	0	0.5
5	6.0	0	2.5
10	12.0	0	5.0
20	24.0	0	10.0

Table 2 contains data points measured in moving state. In Figure 5, we see that this traffic does not appear to be linearly related to the number of vehicles. We suspect that this has to do primarily with deadlock situations when the number of vehicles is high. Otherwise it would mean that for some certain number of vehicles (more than 30), there would be no increase in traffic as more vehicles are added to the system.

To sum up, the traffic generated in different situations is as follows, deduced from the data in Tables 1 and 2:

- Available vehicles and vehicles in deadlock generate on average *1.2 bytes/s* of status messages per vehicle.
- Vehicles processing orders generate on average *17 bytes/s* of status messages per vehicle.
- The server sends *3.2 bytes/s* of command messages per running vehicle.

Table 2: Traffic generated in moving state (avg. bytes/s)

No. of vehicles	Status msg.	Command msg.	Command status msg.
0	0	0	0
1	19.0	3.4	0.2
3	54.8	9.6	0.7
5	92.5	16.8	1.2
8	138.2	27.2	2.2
10	187.8	33.6	3.3
13	211.8	39.8	4.2
20	298.9	52.7	7.5
30	335.0	59.5	10.1

- Each vehicle sends *0.5 bytes/s* of command response status messages in standby state and *0.33 bytes/s* in moving state.

We ran the simulation for different periods of time, varying between 10 minutes and 10 hours. The behavior of the model is rather predictable, as Figure 6 depicts. With the limited set of controllable parameters in the model, patterns in the simulation output are more apparent and repetitive than in output from the real system. An important reason for this is that we cannot take segment lengths, vehicle position and varying order processing time into account in the model. Furthermore, there may also be factors affecting the network traffic that we have not found.

One reason that we cannot say much about the simulation results, is that its inputs do not match the inputs to the real system. In other words, we cannot validate the model

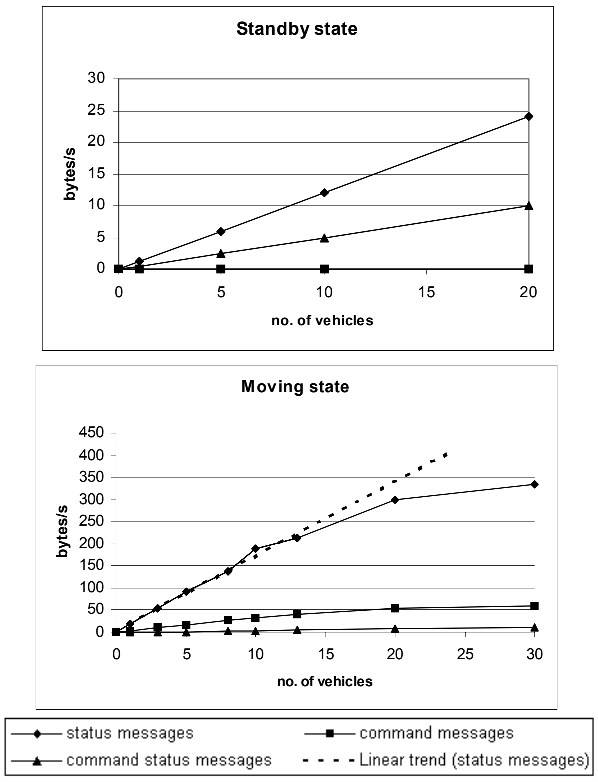


Figure 5: Relation between number of vehicles and the generated traffic.

using the simulation results. Even if we could extract all external inputs to the real system, they would not all apply directly to the model because of the approximations made.

In the simulation output in Figure 6, we see that the average network utilization in the simulation is higher than in the real system (Figure 5). The reason is that the model keeps vehicles busy as the number of vehicles increase, while the real system is not because of the fact that deadlocks and conflicts occur more often there. The lack of variation in the traffic diagram in Figure 6 is an effect of the fixed order processing time, and the fact that vehicles do not enter deadlock until the end of the busy loop.

6.4. Problems with the approach

One fundamental problem with the simulation technique we have focused on, is that it is not possible to distinguish between single "entities" that make up flows in the simulation. An example is the balance of available vehicles and vehicles in use. The time it takes for a vehicle to process an order has to be set to some average time, because the tool does not allow us to associate a random process time with each vehicle. This has to do with the fact that, in continuous simulation, entities are not atomic.

A possible solution in our case would be to let each vehicle be a part of the model instead of being an entity that flows through the model. In such a model, however, the complexity would increase with the number of vehicles. In particular, to change the number of vehicles in the model, one would have to modify the model itself, rather than just one of its parameters.

One of the simulation parameters is the total number of vehicles in the system. The number of vehicles must be chosen carefully, as it has great impact on the efficiency of the system as a whole. In addition, it is not possible to add an arbitrary number of vehicles without taking into consideration the size and complexity of the segment map.

As mentioned, the processing time for an order is set to a fixed value due to limitations in the tool (and simulation technique). In the real system, the processing time depends

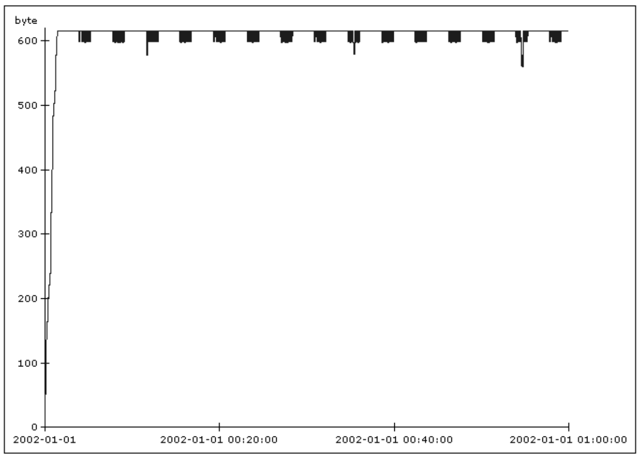


Figure 6: Output from a simulation of 30 vehicles.

on a number of factors, e.g., the vehicle's location, the size of the map, and where the loading stations are

Parameters that have to do with the segment map, such as number of segments and segment lengths, are not included in the model at all. For the same reason as the processing time for an order is fixed, it had not been possible to include other than average values.

In an architecture, the primary entities are components that act together as a whole system. Connections between components can be of the same importance as components, but can also be assumed to simply exist when needed. A reason for this may be that connections can be realized by standardized protocols, e.g., CORBA. In a simulation model like the one we have created, the connections control how data are moved, and components are often merely data generators or data containers, e.g., see the network component in Figure 3. It represents a connection, but is not modeled as a simple connector. Instead, it is a complex unit to show the characteristics it is supposed to have. Thus, the components of the model do not map the components of the architecture very well.

7. Discussion

We have found that the part of the simulation process that was most rewarding was to develop the model. When creating the model, you are forced to reflect over the choices that has to be made in the architecture, resulting in a deepened understanding of the system that helps to identify potential points of concern.

When creating a model of a system, lots of decisions are taken to simplify it in order to speed up the modeling process. A simplification of some system behavior may be valid to make, but if it is erroneous it may as well render the model useless. Therefore, each step in the modeling has to be carefully thought through, something that slows down the entire modeling process.

A model easily becomes colored by the opinions and conceptions of the person that creates the model. Two persons may model the same system differently from each other, which indicates that it is uncertain whether or not a model is correct. Model verification and validation (see chapter 3) are the apparent tools to use here, but it is still inefficient to risk that a model is not objectively constructed. Therefore, we recommend that modeling always should be performed in groups.

While experimenting with the simulation tool, we have found that the ability to simulate a system is a good way to provide feedback to the modeler. It is possible to get a feeling for how the system is going to behave, which is a good way to find out if something has been overlooked in the architecture model. We believe this is independent of the method of simulation that is being used.

While building our experiment model we found that a library of model building blocks would have been of great help. The availability of a standardized way of modeling basic entities such as processes, networks, etc. would both speed up the modeling process and allow modelers to focus on the architecture instead of the modeling.

When simulating a software architecture, the focus can be put on different aspects, e.g., network or CPU utilization. The choice of aspect dictates what in the model that has to be modeled in detail. In our experiment, we chose to look at network utilization, and therefore it is the communication ways in the architecture that have to be specifically detailed. This is noticeable in that communication channels in the model are complex structures rather than simple lines as in an architecture diagram.

8. Conclusions

In this paper we have evaluated the applicability of continuous simulation as a support tool during evaluation of software architectures. Unfortunately, we conclude that continuous simulation does not fit for evaluation of software architectures. There are three reasons that make us come to this conclusion.

First, if continuous simulation is to be used, then we have to use average flow values when we parameterize the model. This makes the model become less dynamic and may have the consequence that the simulation model can be replaced with a static mathematical model.

Second, it is impossible to address unique entities when using continuous simulation. This is not always necessary when simulating flows of information, but if the flows depend on factors that are discrete in their nature, for example vehicles in an AGV system, then continuous simulation is a bad choice.

Third, the process of creating a model for simulation takes considerable time. Since an architecture evaluation generally has to be completed within a limited time, modeling becomes an impractical and uneconomical activity to perform during an evaluation.

We do, however, still believe that an architecture modeling tool that incorporates some simulation functionality could be helpful when designing software architectures. It could for example provide functionality for studying data flow rates between entities in an architecture. Such a tool would preferably be based on combined simulation techniques, because of the need to model discrete factors.

Acknowledgments

We would like to thank Mikael Svahnberg, Blekinge Insitute of Technology, and Henrik Eriksson and Lars Ericsson, Danaher Motion Särö, for valuable discussions, comments, and information about their AGV system. Finally, thanks to Åke Arvidsson at Ericsson AB for valuable simulation advises.

References

[1] Alan A., Pritsker B.: "Principles of Simulation Modeling", in *Handbook of Simulation*, Banks J. (editor), John Wiley & Sons, Inc., pp. 31-51, 1998.

[2] Allen R., Garlan D.: "A Case Study in Architectural Modelling: The AEGIS System," *Proc. 8th Int'l Conf. on Software Specification and Design,* March 1996.

[3] Balci, O.: "Principles and Techniques of Simulation Validation, Verification, and Testing," *Proc. 1995 Winter Simulation Conf.*, pp. 147-154, 1995.

[4] Banks J., Carson II J. S.: "Introduction to Discrete-Event Simulation," *Proc. 1986 Winter Simulation Conf.*, pp. 17-23, December 1986.

[5] Banks J.: "Principles of Simulation", in *Handbook of Simulation*, Banks J. (editor), John Wiley & Sons, Inc., pp. 3-30, 1998. ISBN 0-471-13403-1.

[6] Bosch J.: *Design & Use of Software Architectures*, Pearson Education Limited, ISBN 0-201-67494-7

[7] Davis D. A.: "Modeling AGV Systems," *Proc. 1986 Winter Simulation Conf.*, pp. 568-573, Dec. 1986.

[8] Gannod G., Lutz R.: "An Approach to Architectural Analysis of Productlines," *Proc. 22nd International Conf. on Software Engineering*, pp. 548-557, 2000.

[9] Garlan D., Monroe R., Wile D.: "Acme: An Architecture Description Interchange Language," *Proc. CASCON'97*, November 1997.

[10] Garlan D.: "Software Architecture: A Roadmap", *The Future of Software Engineering, 22nd International Conf. on Software Engineering*, June 2000.

[11] Klingener J. F.: "Programming Combined Discrete-Continuous Simulation Models for Performance," *1996 Winter Simulation Conf.*, pp. 833-839, 1996.

[12] Luckham, D. C.: "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," *DIMACS Partial Order Methods Workshop IV*, Princeton University, July 1996.

[13] Maria A.: "Introduction to Modeling and Simulation," *1997 Winter Simulation Conf.*, pp. 7-13, Dec. 1997.

[14] Richmond B.: *An Introduction to Systems Thinking*, High Performance Systems, Inc., 2001, ISBN 0-9704921-1-1.

[15] Sargent R.: "Validation and Verification of Simulation Models," *Proc. 1999 Winter Simulation Conf.*, pp. 39-48, 1999.

[16] Schriber T. J., Brunner D. T.: "How Discrete-Event Simulation Software Works", in *Handbook of Simulation*, Banks J. (editor), John Wiley & Sons, Inc., pp. 765-811, 1998. ISBN 0-471-13403-1.

[17] Shannon, R. E.: "Introduction to the Art and Science of Simulation," *Proc. 1998 Winter Simulation Conf.*, pp. 389-393, December 1998.

[18] Shaw M., Garlan D.: *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Inc., 1996. ISBN 0-13-182957-2.

[19] Svahnberg M., Mattsson M.: "Conditions and Restrictions for Product Line Generation Migration," *Proc. 4th Int'l Workshop on Product Family Engineering*, LNCS 2290, Springer Verlag, Germany, 2002.

[20] Thesen A., Travis L. E.: "Introduction to Simulation," *1991 Winter Simulation Conf.*, pp. 5-14, Dec. 1991.