# Requirements Monitoring
# in Dynamic Environments

Stephen Fickas
Department of Computer Science
University of Oregon, Eugene OR 97403
*email:* fickas@cs.uoregon.edu

Martin S. Feather
USC / Information Sciences Institute
4676 Admiralty Way, Marina del Rey CA 90292
*email:* feather@isi.edu

### Abstract

*We propose requirements monitoring to aid in the maintenance of systems that reside in dynamic environments.*

*By requirements monitoring we mean the insertion of code into a running system to gather information from which it can be determined whether, and to what degree, that running system is meeting its requirements. Monitoring is a commonly applied technique in support of performance tuning, but the focus therein is primarily on computational performance requirements in short runs of systems. We wish to address systems that operate in a long-lived, ongoing fashion in non-scientific, enterprise applications.*

*We argue that the results of requirements monitoring can be of benefit to the designers, maintainers and users of a system - alerting them when the system is being used in an environment for which it was not designed, and giving them the information they need to direct their redesign of the system.*

*Studies of two commercial systems are used to illustrate and justify our claims.*

## 1: Requirements monitoring in dynamic environments

We focus on requirements engineering issues arising in domains where the environment cannot be counted on to remain static. The general problem is that requirements, and the designs that emerge from those requirements, are typically formulated within the context of an assumed set of resource and operating needs and capabilities. As the environment changes, it may render those assumptions invalid, necessitating the corresponding evolution of the system. This phenomenon is particularly prone to occur in what Lehman has termed “E type systems”, whose installation in some real world domain induces changes in the environment itself, and so leads to altering the system’s own requirements [Lehman 1980].

The two major questions we have been studying are as follows:

i) How can we know when our system needs to be evolved? In particular, how can we carry through to run time the assumptions of resource and operating needs and capabilities made at design time? For example, if we design our system under one set of assumptions about the environment, how can we know when they become invalid once the system is in operation?

ii) Suppose we could detect environment changes that necessitate evolution of our system - how can we use this information to orchestrate this evolution? In the ideal case, we would like this process to be automatic: monitoring information would be consumed by the system itself, which would adjust its own structure or functionality. More prosaically, we may supply this information to the human maintainers of the system, who will thus be aided in their task of evolving their system.

Our approach has been to cast the first question as a problem of *requirements monitoring:* we advocate that as part of the design of a system, requirements monitors be installed to gather and analyze pertinent information about the system’s run-time environment. We formulate specifications of what to monitor so as to gather the information needed to detect divergences from our assumptions that adversely affect adherence to requirements. We address the second question by recording at design time not only the requirements, but also the assumptions comprising the context in which those requirements were formulated, and the compensatory evolutions that we might employ when those assumptions become invalid.

## 2: First case study - license managers

We have studied the above issues using a small but representative problem, a distributed license manager running in an enterprise. The purpose of a license manager is to allow duplicate copies of a piece of software to be used simultaneously by some number of users; the enterprise

Appeared in Proceedings of the IEEE International Conference on Requirements Engineering, 1995

will have purchased a number of 'licenses' for that software, and at any one time up to that many users should be allowed to simultaneously use the software; the vendor, from whom the licenses were purchased, relies upon the license manager to ensure that at any one time the number of simultaneous users does not exceed the number of purchased licences. License managers are of interest to us because there is a wide range of environments in which they reside. The environmental features affecting the license manager include number of potential users, patterns of usage, number of licences purchased, network performance, and available computational resources. More importantly, those environments are often *dynamic* - that is, their features vary over time. The number of potential users may vary, new computational resources replace old ones, etc. Such volatility is inevitable in companies that keep pace with changing economic circumstances and changing technology.

We have studied one license manager in particular - FlexLM, distributed as part of the Solaris software package by Sun. FlexLM has been designed to be applicable in a wide variety of environments. It offers a set of "design parameters" to tune. In essence, the designers of FlexLM anticipated a range of different types of environments in which the system might be deployed, and provided system administrators with some design freedom in setting up the license manager for operation in their particular environments. Thus for our purposes, the 'design task' we focus upon is the selection (and re-selection) of those parameter settings. This frees us from the need to modify program code, which is itself a very difficult task!

We are interested in what happens as changes occur to the environment in which the license manager has been placed. An example is a change in pattern of usage by users - if they become tardy in returning licenses when they no longer need them, and this is causing other users to have to wait a long time to get a license, the administrator may wish to switch to a design in which licenses have a time-bound placed on them (or to decrease the time bound if such a design is already in use).

We will return to the example of license managers after describing more of our general approach.

## 3: General approach - linking requirements, assumptions and evolutions

In general, our approach is to establish the relationships among the following three concepts:

- the overall *requirements*,
- the *assumptions* made about the current state of the environment, and
- the set of remedial *evolutions* available when mismatches develop between assumptions and the current environment.

We propose the use of monitoring to detect the relevant changes to the system's environment. What to monitor for is determined by consideration of the relationships among requirements and assumptions. This yields a specification of monitoring needs. From such a specification, run-time monitoring code (that gathers information and performs analysis) is compiled. Related work on monitoring for debugging and performance tuning provides existing capabilities for such compilation (for a survey of such work, see [Mansouri-Samani & Sloman, 1993]). Note that for debugging or performance tuning the perturbation caused by the insertion of monitoring code threatens to disturb the information gathered and the conclusions drawn from that gathered information, and so must be done with great care. However, what to monitor for is usually obvious, for example, unbalanced loads on multiple processors, or communication bottlenecks. In contrast, for our purposes the perturbation induced by insertion of monitoring is not usually of great concern because it will not usually alter the inferences we draw from the gathered information, whereas the determination of what to monitor is the essence of the problem. Thus we feel confident that once we have developed monitoring specifications (i.e., determined what to monitor for) we can readily apply existing monitoring tools and techniques to create the actual monitoring code. We therefore will focus solely upon the relationships between requirements and assumptions about the environment, to understand how they give rise to monitoring specifications and how the results of monitoring can be applied.

## 4: More on the license manager

The overall requirements of our license manager are as follows:

1. *At any one time, the number of simultaneous users of a piece of software should not exceed the number of licenses purchased for that software.*
2. *Users should not have to wait unduly long for a license.*
3. *No more licenses than are necessary should be purchased.*
4. *Users should find the license manager to be as unobtrusive as possible.*
5. *The running license manager itself should not overly burden the system resources (cpu time, network bandwidth, storage space).*

Note that the vendors of software, and the users of software, have competing interests. For example, users

.

**Table 1:** *Users should not have to wait unduly long for a license*

| Subdivided Requirement | Assumption | Remedy |
|---|---|---|
| Licenses sufficient for user population | User population < k | Purchase more licenses or reduce user population |
| | No more than x% of user population wants to use at once | Purchase more licenses or reduce user population |
| Individual users served licenses fairly | Longest waiting user gets license first | Have license manager maintain queue of waiting users |
| | Users do not hog licenses | Issue time-bounded licenses |
| | | Revoke licenses of current users |
| Users not kept waiting if licenses are available | License manager on reliable platform | Relocate license manager to more reliable platform |
| | | Employ more robust license manager design (backup, majority) |
| | License requests do not become backlogged at license manager | Subdivide license manager & licenses across several platforms |

might like to 'cheat' by violating the first requirement, while vendors might prefer a license manager that made it likely that users would purchase more licenses than strictly necessary. The license manager itself sits in the middle of these competing interests, and our presumption is that the above set of requirements (or something like it) represents a balance deemed fair and acceptable to all concerned.

In most cases, requirement 1 is a 'hard' requirement, ensured by the license manager. There is little purpose in trying to monitor for violations to this requirement, since our monitoring would likely not be any more effective than the license manager itself in detecting violations.

The remaining requirements are typical of 'soft' requirements, which are tricky to design for, particularly in the context of a dynamic environment. They are expressed with varying degrees of precision (e.g., numbers 2, 4 and 5 are stated rather informally). They may be mutually incompatible (e.g., improved satisfaction of 2 through 4 may require consumption of more system resources, thus degrading satisfaction of 5). It is these requirements that induce the greatest need for the kind of monitoring we advocate, and offer the greatest challenge to determine precisely what to monitor for. Because FlexLM's design parameters give us the freedom to tune its installation, we can readily explore a large space of alternative designs that achieve a variety of compromises among these requirements.

To illustrate our approach, we now consider requirement 2 in more detail: *Users should not have to wait unduly long for a license to use a piece of software*. We manually subdivide this requirement into several cases, each of which is a finer-grained requirement. For each subdivided requirement we identify the corresponding assumption(s), and in turn, for each assumption, the corresponding remedy(ies) of how to evolve the design in the case that the assumption is violated. This is shown in table 1. In general, subdivision of requirements is done by following a process closely related to that described in [Dardenne, van Lamsweerde & Fickas, 1993]. A top-level requirement is subdivided and the assumptions behind the resulting sub-requirements are identified, to emerge with assumptions that are candidates for monitoring and remedial action. Generally, this process clarifies the informality present in the initial requirements. The last step is to identify possible remedies to apply when the assumptions are violated; remedies take the form of evolutions to be applied to the system's design.

For example, the initial requirement can be monitored (by watching for a user who is kept waiting longer than some pre-determined time for a license), but has no immediately identifiable assumptions or remedies to take upon detection of violations. In contrast, the above sub-requirements do have clear assumptions underlying them, such as the bound on the user population. Some of the remedies are straightforward, although not necessarily acceptable (e.g., purchase of more licenses will require additional funds, which might not be available). Some depend upon conditions that arise because of the imperfect nature typical of the distributed environments within which most license managers must operate - communication over networks can degrade or fail, individual machines (on which users and/or the license manager itself are running) can become overloaded or fail. For example, we may make an assumption of high reliability of the machine on which the license manager will be located, and, on the basis of this assumption, select a design that will (i) cause the license manager to run on that one machine, and (ii) cause licenses to expire whenever the license manager itself is inoperative (in particular, when it's machine crashes)[1]. Monitoring for violations of this assumption (i.e., downtime of the machine on which the license manager is located) can be used to detect when this has caused users holding licenses to lose the use of them, and waiting users to be unable to get a license. One possible remedy is to switch to a design in which the license manager is replicated across several machines, and a user's license remains valid as long as that user remains in live communication with a majority of those machines. Note, however, that this design is less satisfactory with respect to requirement 5, and so should not be selected without good reason.

As well as gathering information on how the assumptions and requirements are met (or not met) by the current design, monitoring can also be used to answer 'what if' questions about candidate alternative designs. Continuing the preceding example, if the current design is of the manager running on a single machine, we could monitor for how much more reliable it would have been had the manager been subdivided into several incarnations running on separate machines, by monitoring the status of not only the license manager's machine, but also the status of those other machines.

We have experimented with monitoring a simulation of license management, modeling the key concepts of users, licences, etc., and encoding monitoring queries as AI-like daemons that watch for occurrence of those monitoring conditions. This is straightforward to do using our in-house AP5 environment, which provides modeling capabilities together with the ability to declare daemons whose triggers have access to all the information present in the model [Cohen 1989]. Our focus has been the determination of what to monitor for; making the monitoring itself efficient has previously been studied by our colleagues [Liao, 1994].

While our simulation allows us to place monitors in the right places, it is worth noting that FlexLM, itself, does not provide for many of the types of environment monitors we discuss in this section. For example, FlexLM does not provide a "tardiness monitor" that allows us to measure statistics on users keeping a license longer than necessary. Looking more generally at table 1, we require observation of changes to: size of user population, size of license pool, status of platform (running or down?), status of license server users (waiting for license?), etc. FlexLM does not offer broad support for such observations. The next system we look at, Lotus Notes, provides much more detailed "logging" information of events that relate to our assumptions. Given that we don't often need to know immediately violations of our assumptions -- for the purposes of monitoring "soft" requirements, it is not crucial that the system detect violations at once -- we can afford to gather information as it becomes available (logging), and do the analysis opportunistically (during off hours, when connections are restored, etc.).

## 5: Second case study - Lotus Notes

We now turn to a new domain, that of building groupware applications using Lotus Notes. This domain provides several benefits from our point of view:

1. It is a complex problem - developing Lotus Notes applications retains the difficult aspects of FlexLM (distributed application, efficiency issues, security issues) while adding several new concerns (database/groupware issues, mobile clients).
2. It is a problem that is in need of automated support - system administration of Lotus Notes is known to be a hard task. At least part of the difficulty arises from keeping up with a changing operational environment, a problem that we wish to address in our work.
3. Lotus Notes has built-in monitoring tools (FlexLM has no such tools). While these tools are low level, they do provide a foundation to build more sophisticated requirements monitoring tools such as those discussed in this paper.
4. Lotus Notes, in general, is an open system. It runs across platforms (Unix, PC, Mac) and has a well defined API. It is well suited for actual field evaluation (as opposed to strictly laboratory simulation as the case with FlexLM). We have an

1. The latter may seem to indicative of a poor design decision, but in fact admits designs in which licenses are quickly 'retrieved' when users' machines crash, and so supports requirement 4.

interest in testing our work in a semi-realistic environment and Lotus Notes gives us the capability to do so.

For these reasons, we have turned our post-FlexLM attention to Lotus Notes as a study domain. We will describe an example problem in this domain. For readability, we will attempt to parallel our presentation of the previous FlexLM example.

### 5.1: Brief introduction to the Lotus Notes domain

Lotus Notes supports a distributed web of databases shared by users. Each database is associated with a primary server, which is where conceptual changes to the database take place (e.g., a change in the structure or views of a database). However, databases can be replicated across servers. This allows client programs (associated with database users) to choose the server that is most convenient for them to work with.

Lotus Notes supports mobile users through two mechanisms: (1) dial-up service and (2) local databases residing on a laptop. As an example, a mobile user M might perform the following steps:

1. Before going in to the field, download a database D from server A to M's laptop, making a local copy of D on the laptop.
2. In the field, M makes changes to the local D as appropriate.
3. Periodically M phones in to the most convenient server (i.e., server A or any other server that replicates the database D from A) and updates the local version of D (as well as updates the main version of D residing on A and other servers replicating D).

A particular point worth noting is one brought up by replication: how will problems of inconsistency be resolved? In our scenario involving mobile user M, what if M makes changes to the local D (in step 2) that are inconsistent with changes others have made to the main version of D? In step 3, Lotus Notes will find the inconsistencies when M phones in and an attempt is made to bring both local and main versions of D up to date. Lotus Notes handles these inconsistencies by splitting the database D into two versions on server A. The administrator of D (every database has an associated administrator) is alerted to the inconsistent versions, and must manually join the two versions back into one. This joined version is then replicated where necessary.

An interesting conflict is raised by the way Notes handles inconsistency: a mobile user M can selfishly stop phoning in with new updates (to avoid long and onerous transfer times over slow phone lines), and instead dump all changes accumulated over a trip when M gets back to the main office. This takes the burden off of the user M to attempt to work with the most up to date database, and puts the burden on the database administrator to deal with potentially wildly inconsistent database versions once M returns. The conflict among vendors and users in FlexLM is similar in spirit. A broad approach to the role of conflict resolution in requirements engineering is taken up in [Robinson&Fickas, 1994]; we will not touch on it in any depth in this paper.

In summary, there are several key questions in designing a Lotus Notes application:

- How many servers are needed and in what topology?
- Related, what databases reside on what servers, both as primary and replicated databases?
- How often should replication be done for LAN-based servers?
- How often should replication be done for mobile users?
- How much of a database needs to be replicated for specific users or groups of users? (Lotus Notes provides tools for replicating only pieces of a database on a per server or per user basis.)

Taking the perspective raised in this paper, each of these questions has a different answer depending on the current state of the operational environment, e.g., how many users, usage patterns, number of databases, size of databases, mobility necessary, etc.

### 5.2: Applying our method to Lotus Notes

For Lotus Notes applications, we postulate the following as the key top-level requirements:

1. Efficient use of system resources (cpu time, storage space, network bandwidth, etc.).
2. Security - certain documents have access restrictions (e.g., only certain sets of users can read them, and a still smaller subset can write them).
3. Maintainability - efficient use of administrators' time.
4. Speedy user-access to information held within Notes databases.
5. Access to up-to-date information held within Notes databases.

Requirement 2 is a hard requirement, ensured by Lotus Notes. As was the case with FlexLM, there is little purpose in trying to monitor for violations to such hard requirements, since our monitoring would likely not be any

**Table 2:** *From Administrator's Guide*

| To assess... | Use this statistic | Comments |
|---|---|---|
| Load | Server.Trans.Total | Use for monitoring how much the server is in use; if this number is consistently higher than other servers and performance is a problem, you may want to distribute the server's load or add a server. |

**Table 3:** *Access to up-to-date information held within Notes databases*

| Subdivided Requirement | Assumption | Remedy |
|---|---|---|
| % of accesses to out-of-date data < x% | % of out-of-date data < y% | re-align immediately when assumption is violated |
| | proportion of accesses to out-of-data to accesses to all-data < z% | re-align immediately when assumption is violated |
| User warned if data potentially out-of-date | updated-flag propagated immediately on update | explicitly query all replications of database |
| Critical data *must* be up-to-date when queried on server s | updates to data always made on server s | explicitly query all replications of data-base |

more effective than the system itself in detecting violations.

Again, for the purposes of this paper we will focus in detail on (some of) the remaining soft requirements, to show the application of our methodology to produce a linked structure of subdivided requirements, design assumptions and compensatory remedies. It is interesting to observe that the Lotus Notes documentation already uses something very close to this structure to guide administrator activity. For instance, the administrator's guide discusses means of ensuring continued satisfaction of the sub-requirements of requirement 1 (efficient use of system resources). To illustrate, we reproduce a portion of Table 8-11 on page 8-33 of the Lotus Notes Administrator's Guide [Lotus Notes, 1994] (see Table 2 above).

Observe that the *implicit* assumption here is that the load across servers is evenly balanced. This is in support of a sub-requirement that system resources be used efficiently, namely the efficient use of servers. Lotus Notes provides the capability to monitor usage of individual servers. The recommendation here is that the administrator (manually) amalgamates this measure, gathered from servers across the enterprise, to determine if and when load becomes unbalanced. One of the suggested remedies is to redistribute the load. Comparing this to our methodology, it is clear that we would produce something quite similar. We would be more explicit in identification of the assumption being monitored for, and would like to be able to have the system automatically monitor for the composite condition (unbalanced load), as opposed to leaving this to be computed by the administrator. Since efficient use of computational resources has been a continual concern throughout the history of computers, it is hardly surprising that this kind of approach is documented, and supported by the inclusion of appropriate monitoring capabilities (the 'statistics' of the cited table). Our suggestion is that this same approach can be profitably applied to many kinds of requirements, including those often thought of as `soft'. We focus on one of these, next.

Consider requirement 5: *Access to up-to-date information held within Notes databases.* This is a concern in Lotus Notes applications when the database is 'replicated', because the multiple replicas of a single database are not synchronized continually. Instead, changes are allowed to occur on those different replicas, and from time to time they are re-aligned. Of concern to us is the possibility that a user may access out-of-date data from one of these replicated databases. As is typical with the 'idealized' requirements we have formulated, it is unlikely that any possible design will guarantee to satisfy this requirement perfectly. Instead, any given design will be a compromise among the multiple idealized requirements, where the particular compromise is chosen with a set of operating assumptions in mind. Following our methodology, we subdivide requirement 5 into pieces, identify the assumptions underlying each of those pieces, and associate remedial actions to consider applying when those assumptions are violated.Table 3 shows such a partial breakdown of this requirement.

Lotus Notes describes policies of fixed-interval re-alignment of replicated databases. Clearly, the determination of what the interval should be presumes a set of operating conditions that may well change over time. Furthermore, it is hard to characterize the extent to which such a policy will meet the overall requirement of access to up-to-date information. In contrast, our breakdown of the requirement is in terms of (what we postulate to be) typical sub-requirements; because they are refinements of the idealized requirement, it is feasible to design for some of them to be satisfied perfectly. As indicated, a typical design might make certain assumptions about the operating conditions so as to guarantee satisfaction of the corresponding sub-requirement. These assumptions could be monitored by building the appropriate monitoring computations as compositions of the primitive monitoring tools provided by Lotus Notes. Lastly, we indicate possible remedial actions to take when violations of the assumptions are detected. For table 3, these remedial actions have the potential to be automated: Lotus Notes provides an API that makes available low level functions on which to build higher level remedial actions such as shown in column 3.

## 6: Current state

The long range goals of our research are (i) to integrate a requirements methodology with a requirements monitoring framework, and (ii) to automate the "realignment" of the system when monitoring shows a changing operational environment has allowed requirements to be violated. To date, we have carried out the following steps towards these two goals:

1. We have used the requirements refinement methodology of [Dardenne, van Lamsweerde & Fickas, 1993] to manually evolve the ideal requirements of two systems, FlexLM and Lotus Notes, into specific assumptions. We have shown pieces of these requirements evolutions in this paper.
2. We have used the assumptions derived to produce monitor specifications. We have manually implemented these specifications in an AP5 simulation of FlexLM and on top of Lotus Notes monitoring tools.
3. Given detection of requirements violations, we have manually made modifications to the systems to bring them back into conformance.

Migrating each of these manual steps to automated steps can be viewed as crucial *depending on* which research group you talk to: groups doing performance tuning for parallel scientific applications will point to step 2, the implementation of monitoring, as a key concern; groups doing debugging for the same applications will point to step 3, program modification, as a key concern. Our interest is in requirements engineering, and hence, we have come to focus on step 1, the evolution of requirements and the assumptions that are generated as residue. Our studies of FlexLM and Lotus Notes (and systems of their ilk) dictates a focus on step 1, a model of requirements engineering for dynamic environments, while allowing steps 2 and 3 to move to the background. For example, while (i) automating the implementation of requirements monitors from lower level Lotus Notes monitors, and (ii) restructuring a Lotus Notes system given detected failures from monitoring are both interesting problems, neither is on the critical path, we believe, to deriving value from the research we outline here. Step 1 is. Our current effort is directed towards bringing tool support to the requirements engineering of applications in dynamic environments, and the fielded evaluation of those tools in such systems as Lotus Notes.

## 7: Conclusions

Systems such as FlexLM and Lotus Notes are but a small representative example of systems that must operate in dynamic environments. The 'soft' requirements of such systems are challenging to balance in the context of their operating environment, and their designs must necessarily make assumptions about that environment. We have presented case studies demonstrating the promise of monitoring as the means to determine when those assumptions are violated, and whether, as a consequence, requirements are not being met. It is particularly interesting to note that although the initial expression of requirements often lacks formality, requirements in conjunction with assumptions readily suggest easily formalized monitoring specifications.

We have found that our study of changing environments has given us a new perspective on goal-directed requirements acquisition work. Viewing goals as a type of idealized requirements, alternative goal decompositions (sometimes) correspond to viable alternative designs, where selection has been made on the basis of assumptions. In these cases, we provide a different kind of payback to the designer: in return for that designer explicitly recording the idealized requirements, their subdivisions and underlying assumptions, we may generate run-time monitors, which serve to make the resulting system more robust in the face of environmental change. Alternative designs that are recorded become guides for future system modifications needed to work under new environments.

In conclusion, we believe a focus upon requirements of systems that operate in dynamic environments suggests the need for monitoring as a means to guide the appropriate evolution of those systems. Current practice leaves a gap between the support provided to system administrators, and the challenges they face in maintaining their systems in dynamic, evolving environments. Our approach is intended to help bridge that gap.

## 8: References


[Cohen, 1989] D. Cohen. Compiling complex database transition triggers. In *Proceedings, ACM SIGMOD International Conference on the Management of Data, Portland, Oregon.* SIGMOD RECORD 18(2), June 1989.

[Dardenne, van Lamsweerde & Fickas, 1993] A. Dardenne, A. van Lamsweerde and S. Fickas. Goal-directed Requirements Acquisition. *Science of Computer Programming,* 20(1-2):3-50, 1993.

[Lehman, 1980] M.M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. In *Proceedings of the IEEE*, 68: 1060-1076, 1980.

[Liao, 1994] Y. Liao. Efficiently Computing Derived Performance Data. *Automated Software Engineering,* 1(1):11-30, 1994.

[LotusNotes, 1994] Lotus Notes Release 3.1 Administrator's Guide. Lotus Development Corporation, 1994.

[Mansouri-Samani & Sloman, 1993] M. Mansouri-Samani and M. Sloman. Monitoring Distributed Systems (A Survey). Imperial College Research Report No. DOC92/23, Imperial College, Dept. of Computing, 180 Queen's Gate, London SW7 2BZ, UK (revised version published as Chapter 12 of *Network and Distributed Systems Management*, M. Sloman (ed.), Addison Wesley, 1994: 303-347.

[Robinson&Fickas, 1994] Robinson, W., Fickas, S., Conflict Resolution During Requirements Engineering, In *Proceedings of the First International Conference on Requirements Engineering*. IEEE 1994: 206-215.