INT. J. COMPUTER INTEGRATED MANUFACTURING, 1999, VOL. 12, NO. 3, 193–210

A unified framework for the development of automated manufacturing systems supervisory software for the pharmaceutical industry

HENRY Y. K. LAU and K. L. MAK

Abstract. The myth of software development being a 'black art' has long gone. Today, many software engineering methodologies and Computer Aided Software Engineering (CASE) tools are available to streamline the software development process. This paper introduces a unified framework to develop supervisory software for automated manufacturing systems. The framework co-ordinates the efforts put forward by these methods and tools to reduce the cost of software re-configuration, and to enhance the software's quality. The framework addresses the complete software development life-cycle by using an object-oriented approach, to capture and analyse the behavioural, structural and informational aspects of a system. Traceability, completeness and modularity are the principal emphases of the framework. The framework also provides well-defined milestones and guidelines for practising software engineers who undertake software projects for the development of manufacturing systems. The applicability of the framework is illustrated by considering the development of the supervisory control software for an automated chemistry workstation that is commonly employed by the pharmaceutical industry.

1. Introduction

Software has become a major part of today's manufacturing systems controlled using computers. Software development is therefore an inevitable activity for the construction and maintenance of such manufacturing systems, and the ability to re-configure the software without compromising its quality is even more important. Since the flexibility and efficiency of any manufacturing system depend on the supervisory control software, a methodical software engineering approach is essential. A practical automated manufacturing system often involves the management of a large number of physical sub-systems and internal system states, and the making of real-time decisions in response to the occurrences of many possible events. Hence, the method adopted must be able to capture these salient properties.

Software engineering methods such as those of Yourdon (1989), Ward Mellor, Booch (1994), and Booch and Rumbaugh (1995) have been adopted in the design of flexible manufacturing system software (Adiga 1993, Briand and Esteban 1995, Elia and Menga 1994). In addition to these methodologies, Computer Aided Software Engineering (CASE) tools such as System Architect (Popkin 1996), Select OMT (Select 1996), and Rational Rose (Booch and Rumbaugh 1995) are used to automate the development process and to simulate the design (Carrie 1988). These tools have the benefit of saving time in documenting the software process and, to some extent, of providing ideas to the behaviour of the final system.

Most of the existing methods address different phases of software development with different emphases. In particular, the Object-Oriented Design method introduced by Booch (1994) emphasises system design, whereas the Use Case method by Jacobson (1996) provides a comprehensive approach to system analysis and to capturing the system requirements. Moreover, before a particular software engineering method is adopted, it is important to know the type of systems that the method is intended for.

However, it is often not easy to categorise the supervisory software of manufacturing systems into a specific pedigree, e.g. some of these manufacturing systems software have to meet real-time constraints and support some kind of information database simultaneously. Hence, the development of the supervisory software requires efforts from a number of different

Authors: Henry Y. K. Lau and K. L. Mak, Department of Industrial and Manufacturing Systems Engineering, The University of Hong Kong, Pokfulam, Hong Kong, PR China.

0951-192X/99 $12.00

methodologies. Some CASE tools, such as System Architect (Popkin 1996), try to provide software engineers with a set of comprehensive and generic methods with a firm theoretical basis, which, in principle, is able to assist the development of most systems software. Nonetheless, it will be useful to have a framework to unify some of these very powerful methods, and to concretise their interpretations with clear milestones and guidelines.

Indeed, the idea of using frameworks to assist the analysis and design of complex systems is not novel. Similar frameworks, such as the SEI CMM (Paulk 1995) for developing semiconductor related software, have been proposed and adopted in the electronics industry. Such methods have been proved to enhance the quality and productivity of complex integrated devices by formalising the key steps in the development and production processes. Furthermore, commercial CASE tool developers, such as Select Software Tools and Rational Software Corporation, have started to introduce some unified software tools based on user-friendly graphical interfaces (Booch and Rumbaugh 1995, Select 1996).

The present study presents a unified framework that addresses the overall software development life-cycle for the supervisory control software of a typical manufacturing system. The objectives are to:

(a) bring out the salient features of the mainstream object-oriented techniques (Basili *et al.* 1996, Dettmer 1995)
(b) unify them across the stages of software development
(c) generalise the framework for use in manufacturing system software development
(d) illustrate how the framework can be used in practice.

The first part of this paper discusses and presents the philosophy and the major components of the proposed framework, while the second part illustrates the application of the framework through a case study in the pharmaceutical industry.

The principle of the unified framework is discussed in section 2, while section 3 defines the key components of the framework in terms of measurable milestones and processes. Section 4 presents the two principal activities associated with the framework, namely development and verification. Since the framework is designed for the development of complex manufacturing system software, a case study is undertaken in section 5 to demonstrate the applicability of the framework. The development of the supervisory software for a re-configurable automated chemistry workstation (Corkan and Lindsey 1992) that is commonly employed in the pharmaceutical industry is considered. Such automated workstations are complex and interactive in nature, and sometimes involve specific real-time responses and parallel operations. In the case study, the development of the software is illustrated with the presentation of various framework models which specify the different stages of system analysis and design.

2. An object-oriented framework for software development

The philosophy of our framework is to adopt a 'middle-out' approach in which the development process will start top-down, until the key areas of the activities are identified. The framework then considers each area in detail with a view to identifying key entities. These key entities are generalised to form generic building blocks. These building blocks facilitate the framework to work its way upwards and to see how these building blocks can be used to construct the overall system to satisfy the original system requirements. The framework then moves downwards to produce a better approximation to the solution with more software building blocks being identified. This process is repeated at different levels of abstraction, until a set of software building blocks is identified, and these building blocks will implement the complete system software. The unified framework is characterised more specifically by:

- a number of well-defined milestones by which the software life-cycle can be quantified
- two key processes: refinement and verification, which drive the overall software development life-cycle

The four principal milestones of the unified framework are: requirement, analysis, design, and implementation. These milestones are actually models of the system software with different levels of abstraction. In the proposed framework, they are also used as platforms for verifications.

Refinement is a process which drives the forward path of the software development. In this object-oriented framework, the step-wise refinement is performed in the following areas: procedural, control and data. The refinement process transforms the system from one milestone to another, starting from the requirement down to implementation. On the other hand, verification is guided by another set of processes which assist system designers to measure, assure and

correct the refinement processes. The key idea of verification is to show systematically that a refined system description satisfies a more abstract system description. In practice, verification can be performed in different degrees of rigour. It can be a check of consistency by using cross-references, or it can be fully rigorous by using formal mathematics. The proposed framework provides a multitude of formalities in the verification processes so that formality can be enhanced where appropriate.

The framework is based mainly on a number of existing objected-oriented software methods. These include methods used by Jacobson in dealing with early requirement analysis, Booch for class refinement and verification, Ward-Mellor for real-time specification of system processes, and Hoare's Communication Sequential Processes (CSP) (Hoare 1985) for the analysis of parallel processes behaviour. Figure 1 shows the complete unified framework, indicating its milestones and processes.

In addition, the development processes proposed in the framework promote software reuse, enhance traceability, reduce upgrading costs, help to comply with software standards, and, above all, ensure that the software matches the customer's expectations. Furthermore, in the case of the design of safety-critical software, the provision of formalism in system design and implementation which complements conventional analysis will enhance the reliability and correctness of the software. In the following sections the milestones and the development processes of the unified framework are described.

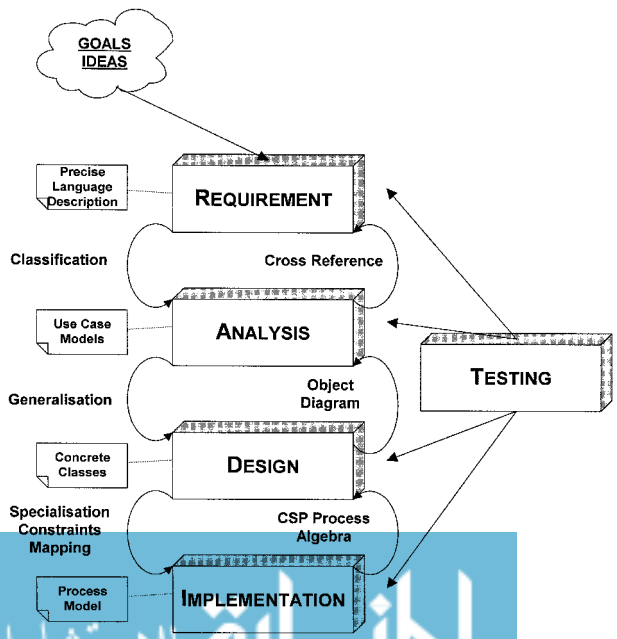

Figure 1. The key components of the Unified Framework.

3. The milestones

The unified software development framework introduces principal milestones which capture the key information developed during the software life-cycle. These milestones quantify the development process, and each milestone is a model of the system with different levels of abstraction in three main aspects: information, behaviour, and presentation. Apart from serving as focal points for the communication of ideas among system engineers and customers, these milestones are the basis for verification. In traditional nomenclature, these four milestones are termed requirement, analysis, design, and implementation models. An additional milestone, the test model, is introduced, which is generated and evolved throughout the software development cycle, and is used to justify the transformation between successive milestones.

3.1. *Requirement model*

This is a document written in a natural language to capture the requirements of the system in a complete, yet precise and unambiguous format. The document is produced on the basis of the functional requirement of the supervisory control of an automated manufacturing system. Apart from system functions, non-functional requirements will also be stated within the requirement model. These may include system performance, quality assurance standards, safety issues, system software and hardware platforms, maintenance criterion, and other ergonomic factors. In all cases, the requirement is the first model to be generated in the software development cycle, to pin down the principal goals.

3.2. *Analysis model*

The principle of the analysis model is to structure the requirement, so that an appropriate platform is created for subsequent system designs. Since the requirement model defines, in a textual format, what the system will do from an operator's point of view, the analysis model formalises these descriptions by using a simple and easily understood notation. Indeed, the analysis model developed describes what the system does, and it is going to be application-oriented, and the actual operating environment into which the system is implemented is not considered. Hence, the main

purpose of the analysis model is to present the problem, and to provide a basis to solve the problem under ideal conditions. As an analysis model is fully problem-oriented, it can easily be developed from a functionality viewpoint. It therefore becomes possible to discuss the model with the users of the system without using implementation jargons.

According to Jacobson (1996), an analysis model specifies what a user or operator expects from the capabilities of the system, and the model gives a conceptual configuration of the system, consisting of representations which denote control, entities and interfaces. With reference to this model, a robust and extensible structure can be developed in subsequent system developments. A complete analysis model should therefore contain sufficient information, so that the total functionality presented in the requirement model is included. Since the model is user-oriented, the analysis model is designed to capture all the events and stimuli, as well as their corresponding responses, which are the major observable entities within a manufacturing system. The notations used by Jacobson are adopted to highlight the following aspects of modelling the system software at an abstract level, to document the information concisely.

3.2.1. *Use case.* A use case is basically a generic description of an entire transaction, or the activities of a system concerned. It specifies the functionality that a system has to offer from an operator's perspective, and defines what should take place within a system. A typical system analysis model may consist of a collection of use cases through which the system organisation and behaviour are completely characterised. When a manufacturing system interacts with its environment, we introduce the concept of an 'actor' into a use case. An actor is a model of an entity that has some kind of dealings with or within the system. Actors are often used to represent the roles that an operator can play within a system, and use cases are used to represent what the operators should be able to do with the system. This type of use case representation, a use case actor representation, is therefore a complete description of the interactions between the system and its environment within the scope of a transaction.

3.2.2. *Use case scenario.* A scenario, on the other hand, is an instance of a use case. It can be a specific function, a reaction to a stimulus from the environment, an internal behaviour, or a system characteristic. For most scenarios, further analysis can reveal a list of events which lead to or make up the scenario. Under this circumstance, an event is a unique and clearly observable situation. A typical event may be an occurrence of external stimulus such as an arrival of a part or a change in system status, e.g. completion of an assembly task. Within an analysis model, the behaviours of a system specified by the use case actor models are further organised into scenarios, with each scenario describing one unique aspect of the system. In this way, the system is progressively analysed and grouped into manageable units that are conducive to subsequent system design.

3.2.3. *Event list for scenarios.* The analysis model is therefore seen as the first attempt in system analysis and design in which the major system components, behaviour and informational details are identified and represented by referring to the observable events. These representations, i.e. use case actor and scenarios, are presented in a combination of textual and diagrammatic forms within our framework in order to enhance the communication of ideas between system designers and customers. Based on these clearly comprehensible use case actor representations and the corresponding event lists, a common language is established, so that any ambiguity, uncertainty, or omission can be removed at this very early stage of the development life-cycle.

3.3. *Design model*

The design model refines the analysis model to facilitate subsequent implementation with an actual implementation environment. The model distributes the behaviour specified in the use case actor models among key entities, and specifies their roles and responsibilities, so that the system's behaviour is realised. In addition, a design model defines explicitly the interfaces between these entities as well as the semantics of the operations. These entities will become the building blocks of the system, and will make up the actual structure of the design model. Despite the fact that these building blocks will later be implemented as executable codes, their actual implementation will be abstracted in the design model. In fact, an entity will often found to be implemented by several classes during the implementation process.

The design model of the framework presents these entities as abstract classes and relationships that reflect their structural organisation. The deliverables in this model are:

- a list of system entities
- a list of abstract classes and their associations with the identified entities

- the hierarchical organisation of abstract classes and the relationships between them.

In the proposed framework, class diagrams (Booch 1994) are used to depict the existence of abstract classes and their relationships in the logical view of a system. These diagrams are used by the design model to capture the structure of the classes that form the system's architecture. A typical class diagram consists of classes and their basic relationships. According to Booch, these relationships between classes include inheritance, 'has', 'using', and association. The first three relationships can be used in physical terms to denote generalisation/specialisation, whole/part, and client/supplier relationships, respectively. The association relationship, being one of the most useful relationships in specifying interactions among entities within a system, denotes a semantic connection between two entities. Associations are often labelled with noun phrases to describe the nature of the relationships. A class may have an association to itself, and it is possible to have more than one association between the same pair of classes. The other three relationships can be seen as refinements of the general association relationship. They are commonly used in the design model to control abstraction during system refinement, and to define specific relationships.

In the design model, a combination of these relationships is used to specify the structural and informational aspects of the system software. In particular, the relationship of inheritance is considered to generalise/specialise classes into base and derived classes. The class hierarchies are formed with the key building blocks (the base classes) clearly identified under these relationships. This allows the generation of the implementation model in which more specialised classes can be formed by means of these base classes. The abstract classes identified within this model are specified by their corresponding methods and attributes. The methods and attributes in a design model are represented at a high level of abstraction with the implementation details not specifically defined. In practice, the abstraction of a design model varies according to the nature of the system concerned, and it is often difficult to define a rigid boundary. Nonetheless, a complete design model must have sufficient abstract classes defined, so that all system scenarios specified in the analysis model can be performed. In order to ensure that this criterion is satisfied, the object diagram analysis may be used, and the use of object diagrams for verification will be described in the test model and the verification process.

3.4. *Implementation model*

Generally speaking, an implementation model consists of detail class specifications, a system process model, and implementation attributes.

The detailed class descriptions which are refined from the design model are defined in the implementation model. Non-functional system requirements, such as constraints, are introduced to specialise the abstract classes. The methods of a class are specified down to pseudo-codes or work instructions and attributes are presented as complete data structures or individual elements with clear data types. Within an implementation model, the classes which are responsible to perform special functions, such as the kinematics of a mechanism or a scheduling algorithm, will combine inputs from other disciplines to complete the specification of the supervisory controller. As such, classes and data structures laid down in an implementation model have strong associations with their physical counterparts. The results are the detailed specifications of all classes and data structures which can be implemented directly.

For supervisory control systems to be implemented on a distributed platform, such as an automated manufacturing system where multi-tasking and parallelism are inherent features, a process model is required. In a process model, classes are grouped into processes or tasks according to the nature of these classes, and to other system attributes, such as the performance criteria, the hardware platform, the operating system and development environment. As regards completeness, all non-functional attributes that define the final form of the system software are included in the implementation model.

3.5. *Test model*

The test model is intended to be used in the verification process to reassure not only the customers, but also the system designers that the product satisfies its initial requirement. The content of the test model is to be developed in conjunction with the construction of other system models. The components of this model will be used at different stages of the development process to ensure that the various milestones generated are correct. A typical test model consists of the following:

3.5.1. *Object diagrams.* Object diagrams represent snapshots in time of an otherwise transitory stream of events over certain configurations of objects. They represent the interactions or structural relationships

that occur among a given set of class instances. Each object diagram is used to represent a system scenario incorporating all the events that occur over the complete life-cycle of that scenario. The object diagrams provide traces of a system's behaviour during the development process. These traces are therefore used to verify completeness and consistency of the system design in the framework.

3.5.2. *System process specifications.* System process specifications are formal specifications in CSP notations which specify parallelism and interactions between processes defined in the implementation model. The use of formal mathematics to construct these specifications allows a fully rigorous proof-of-correctness of specific system behaviour and properties to be undertaken. In most cases, it is impractical to specify fully a complete system in formal mathematics. Nevertheless, the framework provides a methodology to verify the parts of a system formally, which may be of critical importance.

3.5.3. *Software quality assurance test plan and test specification.* Software test plans and test specifications are quality assurance documents which state what is to be tested and how things are to be tested within the scope of system software. The software test plan is usually constructed with the requirement model simultaneously. It lays down all the principle function points of a system that are to be tested. These function points correspond to the use cases identified in the requirement model. A software test specification describes in detail how the final software should be tested against test cases and data. A standard test specification should include unit or component test, integration test, system test, and final acceptance test. Historically, software test plans and specifications are written to streamline the process of software commissioning and hand-over, and are often a prerequisite for the software development to satisfy various commercial standards, such as ISO9001. Nonetheless, they should be seen as important pieces of documentation to system designers for the purpose of verification.

4. The development processes

Having defined the milestones of the unified framework, the other main components are the development processes. There are two key processes: model refinement and verification. Refinement is the underlying process which drives the development life-cycle. Ideally, it starts from the customer requirement. In the proposed our framework, however, the refinement process takes the requirement model as its starting point. Verification on the other hand, is the process by which correctness and consistency are checked after conducting each refinement step. Although this process has traditionally been seen as an auxiliary step by most software engineering methods, we consider it to be as important as refinement. Through experience, the cost of modification of software due to errors in design in the implementation stage is many times higher than that of correction in the analysis stage. It is therefore vital to ensure that consistency, completeness and correctness are maintained during every stage of the development. Hence, verification is a process that is as important as refinement.

4.1. *Model refinement*

The aim of the development process is to produce milestones in a fully traceable and verifiable manner. The refinement process takes the customer requirement, and transforms it systematically into software products. It consists of a set of heuristics to describe the steps of refinement. The refinement process starts from the requirement model, and ends with the implementation model. The following three steps are identified.

4.1.1. *System analysis.* This is the transformation of a requirement model to an analysis model. Since the system analysis aims at generating the analysis model consisting of use cases and scenarios, the main activities involve a close study of the requirement model to identify principal system behaviours, knowledge, interacting sub-systems, and to define the system boundary that separates the supervisory control system with the rest of the system. The following system analysis heuristics summarise the overall activities.

(A1) Construct a system context diagram which defines the supervisory control system boundary and identifies all external sub-systems, including operators, external databases, other hardware and sub-systems.
(A2) Identify all principal actors. These actors include external stimuli and intrinsic system behaviour and knowledge, such as control functionality and database management systems.
(A3) Construct use case. Each use case represents a single system transaction, such as functionality, behaviour and knowledge.
(A4) Identify a list of key scenarios from the use cases and the requirement model.

(A5) Associate actors to each system scenario.
(A6) Analyse each scenario so that the corresponding event list is deduced.

4.1.2. *System design*. System design refines the analysis model to produce a design model. It aims to identify the primary entities of the system from the scenarios detailed in the analysis model. The following heuristics define the logical steps of system classes to identify.

(D1) Identify primary entities. From the system scenarios, logical entities can be identified according to their correspondence with physical objects. In addition, system actors can always be mapped directly as primary entities.
(D2) Consolidate entities or objects. From these entities, abstract classes are identified to represent the functionality of a system. Entities are consolidated into three main categories: interface, informational, and control entities.
(D3) Formation of classes. As entitles or objects are instances of classes, the concept of generalisation/specialisation, whole/part, client/supplier, and association are used to identify the necessary system classes. In addition, the information given in the system object diagrams that are documented in the test model can be used to consolidate the definition of these abstract classes.
(D4) Class diagram representations. Relationships, such as inheritance, aggregation, etc., among classes are represented in the form of class diagrams for easy communications.

4.1.3. *Implementation*. Implementation transforms a design model into an implementation model. Here, abstract system classes are refined to detailed class descriptions with the introduction of constraints. Adaptation is made to cater for the implementation environment, such as operating systems, languages, architecture of the computer systems, third party software ,and hardware, etc. Each method defined in a class is detailed to include algorithms, logic descriptions, etc., and attributes are given in concrete data types. The goal is to produce a detailed system model to generate software directly. The heuristics for system implementation are as follows.

(I1) Integrate system constraints with classes. Abstract classes are refined and specialised according to system functionality and constraints. The methods and attributes in each class are specified in detailed pseudo-codes.
(I2) Group classes into physical processes. By following a similar concept as in system design, classes can be classified generally under three main categories of process: informational entity, control, and interface.

To expand on heuristic (I2), an interface process is one having direct interaction with the external environment, e.g. a communication device driver of a part handling robot. Classes such as the serial communication classes and device driver classes can be grouped as interface processes. An informational entity process is one which implements information within a system. This may be a class which encapsulates data structure or a relational database that holds sequences of assembly operations. Control processes are processes which implement computation, logic controls and state transitions. It is common that these processes are coupled, and that they interact with one another in a synchronous or asynchronous manner.

4.2. *Model verification*

Model verification is to ensure that the refinement process is correct in connection with completeness, consistency and functionality. Although the level of verification often depends on practical factors such as time and cost, the proposed framework addresses the full verification process for the entire life-cycle of software development. By the same token, it is necessary to demonstrate that a complementary verification process must exist in each refinement step. Table 1 summarises the verification processes adopted by the framework.

4.2.1. *Cross-reference*. To establish the fact that an analysis model specifies the requirement completely, the framework uses a matrix to cross-reference function points identified in an analysis model with those described in the requirement model. Since the requirement model is usually written in a natural language, a formal verification is often impractical. A matrix is the most direct and efficient way to match and relate the two models, to ensure continuity and consistency, and to avoid any omissions. As an example,

Table 1. The corresponding verification processes.

Refinement	Verification
System analysis	Cross-reference
System design	Object diagram
Implementation	CSP modelling

a typical cross-reference matrix may include the following entries:

- system functional and non-functional specification headings
- corresponding section numbers from the requirement model
- corresponding analysis model headings
- corresponding section numbers from the analysis model
- corresponding analysis model scenarios.

4.2.2. *Object diagram analysis.* As described in the system test model, object diagrams are used to represent the views of the object structure of a system. In the verification process, object diagrams can be used to indicate the semantics of scenarios via traces of events and operations. One of the key criteria to accept a design in terms of its completeness is to demonstrate by using object diagrams that all system scenarios can be performed by using the classes defined in the design model. Each scenario should correspond to an object diagram to depict a subset of system behaviours. With the production of a complete set of object diagrams to represent all system functionalities, the correctness of the design model can be deduced. In addition, class and object diagrams can provide the necessary information to a number of design metrics, including the total number of classes reused, total number of stimuli sent, and also the number, width and height of the inheritance hierarchies to measure the quality of system designs.

4.2.3. *CSP modelling.* The framework uses the formal mathematics CSP to express and reason about concurrent processes. CSP offers a succinct mathematical notation to describe processes and a way to control the abstraction level of these descriptions. Processes can be structured by using combinators for parallel and sequential compositions, communication, etc. Over the years, a number of practical case studies have been carried out by using CSP as a formal specification method. These include the specification of aircraft engines (Jackson 1989), traffic control systems (Lau 1990), autonomous guided vehicles (Stamper 1990) and reliable network protocols (Hindrey and Jarvis 1995). Further discussion of the semantics of CSP may be found in Hoare (1985).

When a system design is implemented, classes are grouped/partitioned into processes. In order to allow a full understanding and prediction of the behaviour of these abstract implementations, CSP is used to formalise these process descriptions. The formal process specifications allow mathematical reasoning such as process algebra to be performed, to verify properties such as interactions, communication, synchronisation, and deadlock freedom. The deductions from this mathematical reasoning are then checked against the detailed event list for each scenario. A correct implementation must entail the ability to perform all system scenarios for the processes given in an implementation model. As for each scenario, the sequences of events deduced from the CSP verification should match with those originally specified in the design model.

5. Development of pharmaceutical software with the unified framework: a case study

5.1. *Supervisory software for an automated chemistry workstation*

An automated chemistry workstation (ACWS) is a complex system which is designed to maximise the productivity of chemical reactions for the manufacturing of drugs (Corkan and Lindsey 1992). These workstations have emerged to be one of the major automated systems (Chodosh *et al.* 1986, Kramer and Fuchs 1986, Fujita *et al.* 1990) employed in the pharmaceutical industry. Although the hardware and software designs for the workstation are complementary, the focus of this paper is on the development of the software. In this section, the unified framework is applied to the analysis and design of the supervisory control software for an ACWS.

A typical automated chemistry workstation comprises a robotics liquid handling manipulator equipped for solvent addition; liquid sampling and reagent transfer; an individually stirred multi-vessel variable temperature reaction station; a dilution station; a reagent storage unit; and an ultraviolet-visible absorption spectrometer for the chemical analysis. Typically, the transfer of liquid samples is accomplished via electric syringe pumps. These pumps and the five subsystems are interfaced to a central controlling computer via RS-485 links and digital I/O controls. Within this computer, a piece of supervisory software is executed to co-ordinate all the equipment, to interface with the operators, and to maintain a log of system status and experimental details. The organisation of the ACWS is shown in the system context diagram (figure 2).

As in any real system, design constraints also exist in this case. One of the key design criteria for the ACWS is to achieve maximum parallelism. In addition, the hardware for the supervisory software is a single board computer running a real-time multi-tasking operating system. Moreover, the implementation should take

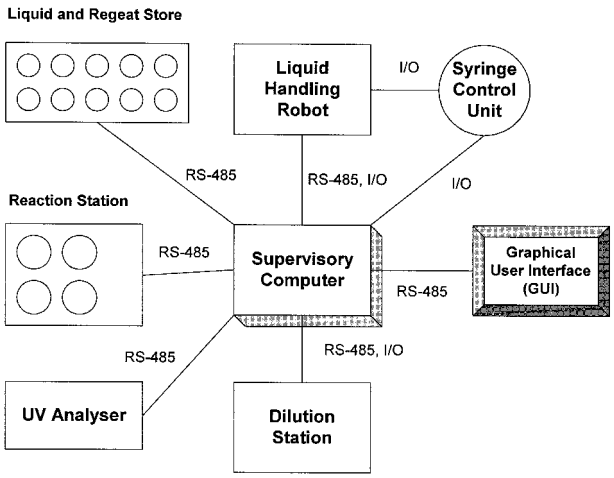

Figure 2. System context diagram for the automated chemistry workstation.

advantage of the distributed hardware sub-systems with embedded processors, to enable various mechanical operations to be performed at the same time.

5.2. *System analysis for the ACWS supervisory software*

Assume that the requirement model previously prepared between the customers and the system engineers is available, and that an extract from the requirement model is as follows.

> 'The automated chemistry workstation is to be operated by trained chemists who shall operate the workstation via a graphical user interface. Experimental runs are entered via the graphical user interface (GUI), and stored locally within the supervisory computer. Once the instruction to perform the experiments has been entered, the workstation shall schedule, manage, and start the experiments in an autonomous manner.
>
> A typical experiment involves the Liquid Handling Robot transferring all required reagents from the Liquid Store to a specific reaction vessel by using the electric syringes. Some reagents may require dilution by using the Dilution Station before being transferred to the reaction vessel. Once a reaction has started, the Liquid Handling Robot samples the reactant periodically, and the system analyses the samples using the UV Analyser. More than one experiment is scheduled at any one time if possible. Experimental results are updated on the computer terminal, and all the results must be stored for further analysis. The system must be able to log and report all occurrences of warnings and errors.'

In view of this requirement, system analysis is performed to produce the analysis model for the ACWS supervisory software. From the system context diagram (figure 2), principal actors are identified according to heuristic (A1). Table 2 presents these actors.

With these actors, the system use case diagram is constructed and is illustrated in figure 3. From the requirement model, a list of representative scenarios is identified, and is given in table 3. These scenarios describe the system behaviours, characteristics, and functions. It can be seen that, for a typical distributed system, each active external sub-system which communicates with the supervisory controller in duplex is assigned as a unique actor. Other passive external devices only acted on by the supervisory controller, such as the Dilution Station and the Liquid Store, are identified as 'passive' entities. Apart from responding to demands, these external actors are the sources of stimuli which drive the system. In the case of the ACWS, two key internal features have to be modelled at this stage of the analysis. As the ACWS is an autonomous system, system functions are also performed without external stimuli. Hence, the Workstation Manager is introduced to capture all the internal behaviours, including the control of user interface, scheduling, and the co-ordination of the sub-systems. The other internal actor which is of equal importance is the Database. Since one of the requirements for the workstation is to be able to retain the experimental definitions and to store and retrieve all the results produced, it is necessary for the ACWS to maintain and manage all the information. The internal actor Database Manager is an abstract description of the system's knowledge encapsulating both informational and operational details of the database.

Following the identification of scenarios and actors, the associations between them are made. These associations can be represented graphically in the form of use case actor scenario diagrams or simply in a tabular format. Table 4 gives the associations between actors and their corresponding scenarios.

Finally, in the system analysis phase, system behaviour is extracted from the requirement model to expand each scenario into a list of events or operations. To illustrate the expansion of the event list, the Prepare Experiment scenario will be analysed.

The Prepare Experiment scenario involves the preparation of a defined experiment so that it is ready for initiation. The Liquid Handling Robot is used to transfer reagents from the Liquid Store to the specific reaction vessel in the reaction station. Some reagents are assumed to require dilution, and the reaction profile is finally loaded into the reactor controller. The steps involved in this scenario are detailed in table 5.

Table 2. Principal actors of the workstation.

Actors	External/ internal	Remarks
Chemist	external	A typical operator.
Database Manager	internal	Informational feature of the supervisory software.
Workstation Manager	internal	Key functionality of the supervisory software, responsible for co-ordination of hardware, scheduling experiments, managing experimental runs and controls of the user interface.
Liquid Handling Robot	external	The robot manipulator responsible for transferring liquids such as reagents using the electric syringe to various hardware modules; it constantly informs the supervisory controller with information such as job progress, current syringe locations and status.
Reaction Station	external	The individually stirred multi-vessel variable temperature reaction station.
UV Analyser	external	The ultraviolet-visible absorption spectrometer for analysing the composition of the reaction content.

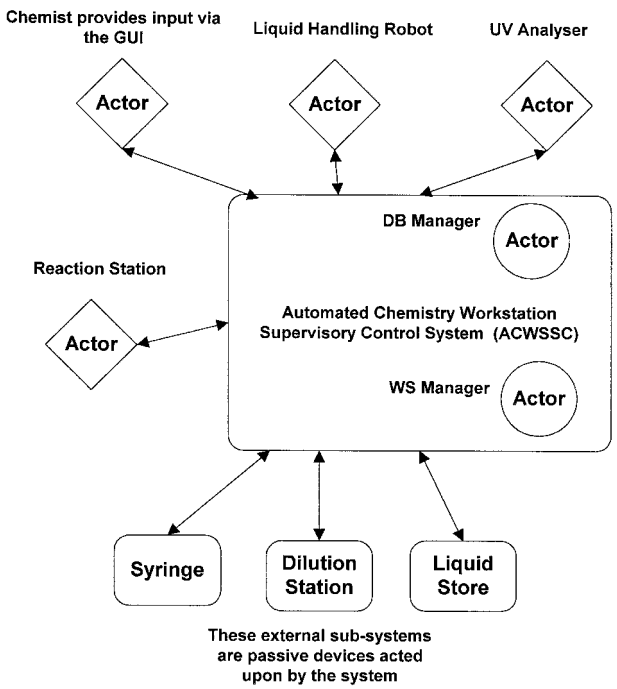

Figure 3. System Use Case diagram.

Table 3. Major system scenarios.

No.	Scenario
1	System start-up
2	System shutdown
3	Define experiment
4	Review experimental result
5	Produce experimental reports
6	Backup system configurations
7	Schedule experiments
8	Handle errors and warnings
9	Periodic system health check
10	Prepare experiment
11	Sample reactant for analysis
12	Monitor and control reaction profile (time, temperature, amount of mixing)
13	Perform UV analysis
14	Generate analysis results

5.3. *System design: the formation of abstract ACWS classes*

The goal of system design is to define all the essential classes for the supervisory software of the ACWS. By following rules (D1) to (D4), the scenarios of the system presented in the analysis model are examined, and the primary entities are identified. According to rule (D1), the entities that correspond to system actors and major sub-systems are given in table 6. Table 7 illustrates the decomposition of the Prepare Experiment scenario for the identification of other system entities.

Having identified the primary entities for actors and scenarios, a consolidated entity list is produced for the system (table 8). In this table, abstract classes that correspond directly to these entities are introduced. However, for actual implementation, the software may contain many more classes than those introduced in table 8. These extra classes include utility classes, container classes, and third party library classes which depend heavily on the chosen development platform and the operating system.

In the ACWS supervisory software, all system entities being identified can be categorised into four key functional blocks according to their behaviours, informational details and responsibilities. These functional blocks define the architecture of the software, and they are: the user interface, manager, virtual device layer, and physical device layer.

Within system design, other primary classes can be identified from the consolidated entities list (table 8) and the corresponding object diagrams in the test model. These classes are defined in terms of their external interface and visibility. To complete the system design process, the relationships between classes are

Table 4. Association of actors and scenarios.

Actor's scenario	Scenario name	Description	No.
Chemist scenarios	System start-up	Handle start up of the ACWS.	1
	System shut down	Handle chemist's request to shut down the system.	2
	Define experiment	Handle experiment definitions by a chemist via the system GUI.	3
	Review experiment result	Handles the display of selected experimental result by a chemist via the system GUI.	4
Liquid Handling Robot scenarios	Prepare experiment	Handles the transfer of reagents from the liquid store to the reaction vessel. Reagents are transferred using the syringes and reagents are diluted using the dilution stations if required by the experimental definitions.	10
	Sample reactant for analysis	Handles the transfer of reactant from the reaction vessel to the UV analyser. Reactant is transferred using the syringes by the Liquid Handling Robot.	11
Reaction Station scenario	Monitor and control reaction profile	Handles control and monitoring of experimental profiles including temperature and experiment time	12
UV Analyser scenarios	Perform UV analysis	Handles analysis of reactant with the UV Analyser.	13
	Generate analysis results	Handles post-processing of the analysis result for system storage.	14
DB Manager scenarios	Review experimental result	Handles the retrieval of experimental results from the system database ready for examination from the GUI	4
	Product experiment reports	Handles collection of experimental results, organisation and processing of results to produce experiment reports.	5
WS Manager scenarios	Backup system configuration	Handles the periodic backup of the system configuration. Store the current experimental profile.	6
	Schedule experiments	Handles the scheduling and sequencing of different experiments such that more than one reaction and analysis can be placed at the same time.	7
	Handle errors and warrnings	Handles the logging and reporting of system errors and warnings.	8
	Periodic system health check	Handles periodic health checks to all external devices connected to the supervisory computer.	9

Table 5. The key steps for the Prepare Experiment scenario.

Step	Key events	Description
1	Get experiment detail	Obtain definition of the experiment from the system database. The definition of an experiment is entered via the GUI by a chemist. It is assumed that the specified experiment has already been defined.
2	Activate Liquid Handling Robot	Initialise the Liquid Handling Robot in preparation for liquid transfer. The robot picks up a fresh syringe for every new reagent to be transferred.
3	Get reagent from the Liquid Store	The Liquid Handling Robot uses the syringe to aspirate the selected reagent from the liquid store. This step plus steps 4 and 5 are repeated for all reagents required for the experiment.
4	Dilute reagent	The Liquid Handling Robot transfers the reagent that requires dilution to the dilution station for dilution. The dilution station will dilute the reagent to its required concentration. If the reagent does not require dilution, this step will not be performed.
5	Transfer reagent to reaction vessel	The reagent (diluted or in its original concentration) is dispensed to the assigned reaction vessel.
6	Set-up reaction profile	The experiment profile is down loaded to the reaction vessel controller.
7	Update GUI	The user interface display is updated.

Table 6. Entities identified from the system use case.

Entities	Description
GUI interface	Identified from the 'Chemist' actor. Handles all chemist presentation and inputs.
Liquid Handling Robot Interface (LHR Interface)	Identified from the 'Liquid Handling Robot' actor. Handles all interaction with the external robot control system.
Reactor Interface	Identified from the 'Reaction Station' actor. Handles all interaction with the external reaction vessel controller.
Analyser Interface	Identified from the 'UV Analyser' actor. Handles all interaction and communication with the UV Analyser.
Database Manager (DB Manager)	Identified from the 'Database Manager' control entity. Manipulates, manages and stores all system data such as experiment definitions, results and progress of experiments.
Workstation Manager (WS Manager)	Identified from the 'Workstation Manager' control entity. Controls and manages system operation including periodic events.
Liquid Store Interface (LS Interface)	Identified from the 'Liquid Store' sub-system entity. Handles all controls of the physical liquid store.
Dilution Station Interface (DS Interface)	Identified from the 'Dilution Station' sub-system entity. Handles all interactions with the dilution station.
Syringe Interface	Identified from the 'Syringe' sub-system entity. Handles the actions of the electric syringes, namely aspiration and dispensing of reagents.

Table 7. Entities identified from the Prepare Experiment scenario.

Entities	Identified in steps	Details
DB Manager	1	Handles storage and retrieval of specific experimental data.
LHR Interface	2, 3, 5	Handles all interactions with the Liquid Handling Robot.
Syringe Interface	2, 4, 5	Handles the operation of th electric syringe.
LS Interface	3	Handles all controls to the Liquid Store.
DS Interface	4	Handles all controls to the Dilution Store.
Reactor Interface	5	Handles all interactions with the Reaction Vessel controller.
Reactor Details	6	Handles storage of reaction profiles settings for each reaction vessels.
GUI Interface	7	Handles the presentation of system information to a chemist.

identified from the scenarios and the list of consolidated entities. The idea of generalisation/specialisation is applied to classes which belong to the same functional blocks. In the case of the ACWS, the physical device layer responsible for the interfacing between sub-systems consists of a number of communication classes which can be organised into an inheritance hierarchy. The generalised *comm_port* class is introduced as a base class to build other more specialised classes, such as the *LHR_Device* and *Reactor_Device* classes. In addition, this *comm_port* class itself is a container class for the two major hardware dependent classes, i.e. the *RS485_Port* and *IO_Port* classes. Within the virtual device layer, container classes such as the *LHR* class are introduced from the idea of the whole/part relationship. These container classes encapsulate the informational (e.g. *LHR_Data*), functional (e.g. *LHR_Device*), and structural (e.g. *Syringe_Device*) details in order to enhance modularity. All these relationships can be documented concisely using class diagrams. Figure 4 shows a top level inheritance hierarchy of the physical device layer.

5.4. *Implementation: a process model for the liquid handling sub-system*

During implementation, the actual software architecture is defined by considering the constraints and the physical details given by the system attributes. These constraints are introduced to further specialise classes, so that the classes can be implemented directly by using an object-oriented language such as C++.

Having detailed all the classes, a final step in system implementation is to construct a process model for the ACWS. This model includes the partition or grouping of classes into processes. In the case of the ACWS software, a multi-tasking operating system is used, and classes are grouped into tasks under similar groupings as the four principal functional blocks: user interface,

Table 8. Consolidated system entities.* This may consist of a hierarchy of classes which depend on operating system and development environment.

Entities	Corresponding classes	Description	Functional block
GUI Interface	GUI_Class*	Handles all chemist inputs and presentation.	User interface
LHR Interface	LHR_Device	Handles all interaction with the LH Robot.	Physical Device Layer
Reactor Interface	Reactor_Device	Handles all interactionrs with the Reaction Vessel controller.	Physical Device Layer
Analyser Interface	Analyser_Device	Handles all interaction with the external.	Physical Device Layer
LS Interface	LS_Device	Handles all controls to the Liquid Store.	Physical Device Layer
DS Interface	DS_Device	Handles all controls to the Dilution Station.	Physical Device Layer
Syringe Interface	Syringe_Device	Handles the operations of the electric syringe.	Physical Device Layer
RS485 Interface	RS485_Port	Handles all communication with a RS 485 port.	Physical Device Layer
I/O Interface	IO_Port	Handles all control to the physical digital I/O port.	Physical Device Layer
LHR Details	LHR_Data	Handles storage and retrieval of operational data of the LH Robot.	Virtual Device Layer
Reactor Details	Reactor_Data	Handles storage of reaction profiles for each reaction vessels.	Virtual Device Layer
Analyser Details	Analyser_Data	Handles storage and retrieval of working data and result of the UV Analyser.	Virtual Device Layer
LS Details	LS_Data	Handles storage of Liquid Store data.	Virtual Device Layer
DS Details	DS_Data	Handles the storage of Dilution Station data.	Virtual Device Layer
Filing System	File	Handles output to the operating system filing system and maintains log files.	Manager
DB Manager	DB_Manager	Handles storage and retrieval of all experimental information.	Manager
WS Manager	WS_Manager	Handles system control as well as containing, controlling and managing the virtual device layer.	Manager

Table 9. Association between scenario events and object diagram steps for the Prepare Experiment scenario.

Step	Key events	Object diagram steps
1	Get experimental detail	1, 2, 3, 4,
2	Activate Liquid Handling Robot	5, 6, 7
3	Get reagent from the Liquid Store	8, 9, 10, 11
4	Dilute reagent	12, 13, 14, 15
5	Transfer reagent to reaction vessel	16, 17, 18, 19, 20
6	Set-up reaction profile	21, 22, 23
7	Update GUI	24

manager, virtual and physical devices. More specifically, each specialised device class is assigned as an independent task, whereas the system database and supervisory control belonging to the manager function block are assigned as two different tasks. The user interface, being a self-contained unit, is assigned as another independent task. In figure 5, the process model of the ACWS software also includes two other tasks, namely, the *Utility* and *Database*. The *Utility* task is introduced to streamline the passing of data around the system. It implements the *Msg* class and *SysInfo* class. These classes contain a number of generic message handling methods used by other system classes to relay system-wide information. The *Database* task implements the system *Database* class and other common data structures. Here, the *Database* class is primarily responsible for the storage of experimental details.

5.5. *Verification issues*

The pharmaceutical industry is a highly regulated industry and has put in a concerted effort to define a computer system life-cycle in terms of the validation process (Liscouski 1995). The correct and proven operation of a computer controlled pharmaceutical system is therefore one of the prime factors in determining the acceptance of such system. Various guidelines, including the PMA's Validation Life Cycle (PMA 1986), which defines the verification steps necessary for system development, are set up in the industry to regulate and assure the quality of these computer-based pharmaceutical systems and software. When comparing these guidelines with the proposed framework with regard to system testing and verification for software, the framework has indeed aligned with these guidelines in terms of the provision of a staged development and a built-in quality assurance programme. Within the framework, the milestones provide

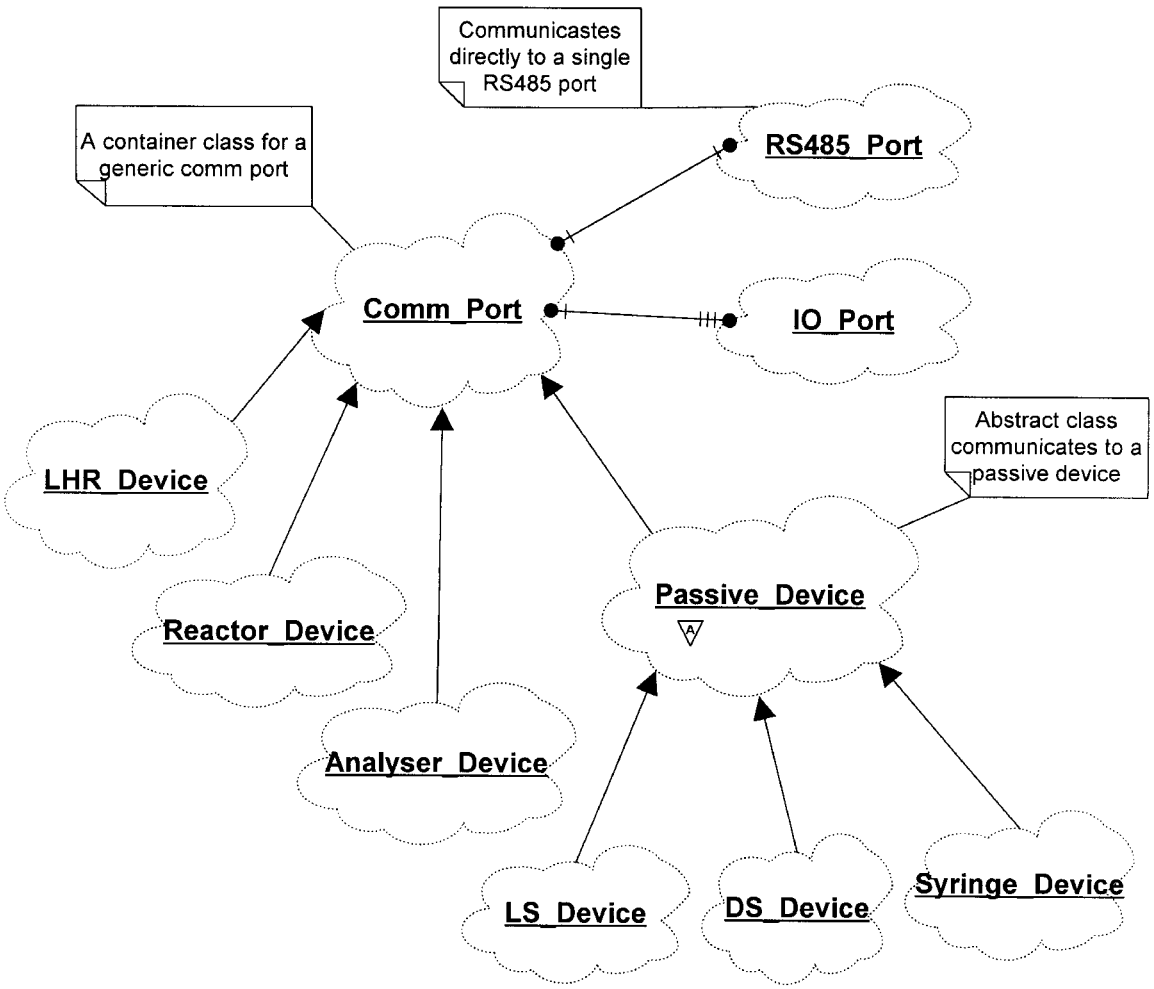

Figure 4. Top level inheritance hierarchy of the ACWS physical device layer.

a means to measure the progress of the development, whereas the test model developed along the entire software development process provides an integrated means of assuring the quality of the software produced. These reinforced the applicability of the framework in developing software for the pharmaceutical industry.

In this section, the verification of the system analysis and design is illustrated through object diagram analysis and reasoning using the CSP specifications of the high level system tasks defined in the system process model (figure 5). Figure 6 is the object diagram for the Prepare Experiment scenario. In this figure, objects are constructed using the instances of those classes identified in the system design. By tracing through the events depicted in the object diagram, and matching these with the scenario steps defined in table 5, the classes so identified are sufficient to perform the entire Prepare Experiment scenario (table 9). By employing this principle, the complete system design can be checked by constructing object diagrams for all system scenarios. In addition to verification, object diagrams can assist the identification of methods and attributes for each class.

Finally, to verify the interactions between tasks, high level CSP specifications are constructed, and the reasoning techniques can be used. The properties that this verification process focuses on are parallelism and synchronisation between the co-operating tasks. Correct functioning of the overall system depends on the way the tasks communicate with one another. Here, we use the same scenario, the Prepare Experiment scenario described previously, and specify each relevant task using CSP. All specified tasks are put in parallel operation in the CSP model to allow for maximum parallelism which is one of the system design criteria. These CSP processes are reduced by using process algebra, and deductions are made in terms of event sequences and parallelism. In this paper, the tasks are

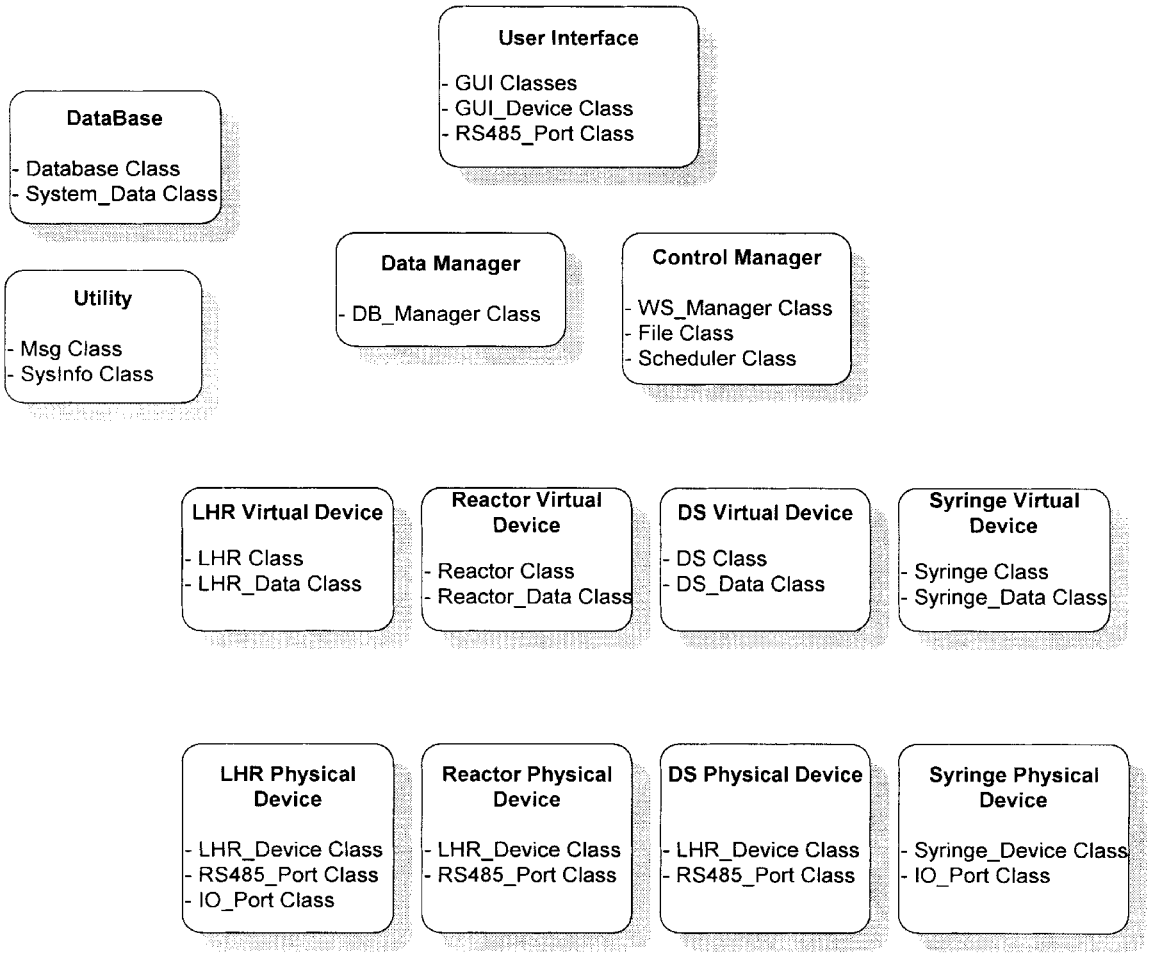

Figure 5. High process model of the ACWS supervisory software.

specified at a high level for the purpose of illustration. Nonetheless, there is no limit on how much detail system the designers may introduce into a particular CSP specification. Figure 7 presents the CSP specifications for each relevant task and the overall system specification, i.e. the ACWS process. The last equation shows that the observable events: *get_expt*, *get_rgt*, *asp_rgt*, *dilute*, *dsp_rgt*, *load_profile*, and *display* match the sequence of scenario steps defined in table 5.

6. Conclusion

The unified framework presented in this paper attempts to introduce some degree of uniformity, and provides a systematic approach to engineer complex supervisory software for manufacturing systems. The provision of milestones makes the development more quantifiable, and the refinement processes suggested a number of concrete heuristics to guide software development. Verification, together with the information defined in the test model, allows each milestone to be checked against its predecessor for correctness, consistency, and completeness. Although the framework provides a full verification path beginning from system analysis to implementation, it does not dictate how far the verification process should proceed. Nonetheless, the framework provides a spectrum of approaches to address the task of verification in order to serve the wide diversity of manufacturing systems. System designers may decide to use only cross-references for low risk systems, or go to the other extreme of formal specification of every individual process in CSP for supervisory software that controls critical operations. In addition to these advantages, the framework has a firm object-oriented basis in which modularity and reusability are the inherited features. These features facilitate flexible and easy system re-configurations due to changes in requirements, which are common cases in modern automated manufacturing systems.

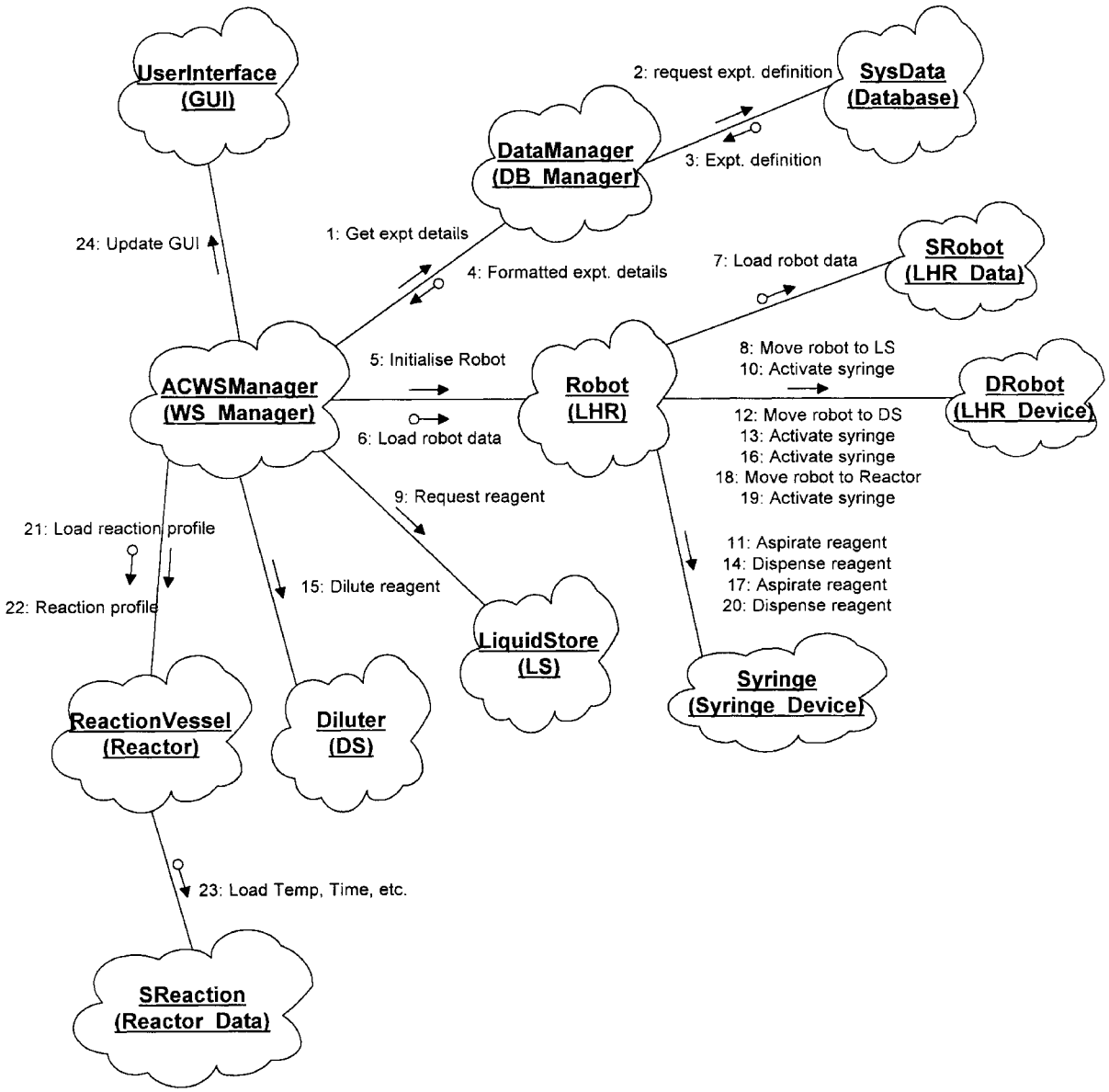

Figure 6. Object diagram for the Prepare Experiment scenario.

With regard to the development of supervisory software for the pharmaceutical industry, it is shown in the case study that the system models generated in the development provide clear, precise and easily understood means for both engineers and customers to work with. In the context of verification, the framework requires the test model to be built in conjunction with the other system models, and for each step in system development, a complementary verification step is to be taken. This approach will enhance the conformance of the software produced, and provide an avenue to prove the correctness of the software with respect to its original requirement. With these features in place, the framework has made it possible for manufacturing system engineers to address positively some of these issues, including quality assurance, correctness, and reliability in the development of pharmaceutical software.

Indeed, the framework has been applied to develop supervisory software for a number of complex manufacturing systems, and improvements can be noted in quality, traceability, maintainability, and documentation. Each of these software projects has a relatively low number of internal change requests, which indicates that the number of errors made during the system development has been reduced. In addition, these projects have successfully passed through software

$$User\_Interface = (get\_ip \mid display) \rightarrow User\_Interface$$

$$Data\_Manager = (get\_expt \mid report) \rightarrow Data\_Manager$$

$$Control\_Manager = get\_expt \rightarrow (get\_rgt \rightarrow asp\_rgt \rightarrow dsp\_rgt \mid get\_rgt \rightarrow asp\_rgt \rightarrow dilute \rightarrow dsp\_rgt \mid system) \rightarrow load\_profile \rightarrow display \rightarrow Control\_Manager$$

$$LHR\_Device = (move \mid asp\_rgt \mid dsp\_rgt \mid status) \rightarrow LHR\_Device$$

$$Reactor\_Device = (load\_profile \mid control\_expt \mid status) \rightarrow Reactor\_Device$$

$$DS\_Device = dilute \rightarrow DS\_Device$$

$$LS\_Device = get\_rgt \rightarrow LS\_Device$$

$$ACWS = (User\_Interface \parallel Control\_Manager \parallel Data\_Manager \parallel LHR\_Device \parallel Reactor\_Device \parallel DS\_Device \parallel LS\_Device)$$

$$ACWS = get\_expt \rightarrow (get\_rgt \rightarrow ((asp\_rgt \rightarrow dilute \rightarrow dsp\_rgt) \mid (asp\_rgt \rightarrow dsp\_rgt)) \rightarrow load\_profile \rightarrow display \rightarrow ACWS$$

where *get_expt* : get experimental details
get_rgt : request reagent
asp_rgt : aspirate reagent
dsp_rgt : dispense reagent
load_profile : load reaction profile
display : update GUI
dilute : dilute reagent

Figure 7. High level CSP specifications for the ACWS tasks with respect to the Prepare Experiment scenario.

audits for standards such as the ISO 9001 and TickIT. However, it is important to understand the ultimate aim of the unified framework, which is to address the analysis of the architecture of software, the organisation of modules, and also the completeness and consistency of the overall design. As such, it is not a tool for the design of specific algorithms, such as kinematics or scheduling solutions, and the ultimate correctness of the software produced depends on the correctness of its original requirement specification. Furthermore, the development, and thus application of a fully unified framework is generally considered to be at its infancy within the community of manufacturing system software businesses. There is room for future improvement in numerous areas, including a unified notation for specification of manufacturing systems, automated tools to assist the transformation between each model and verification, the introduction of formal mathematics to system analysis and, the extension of the framework to a system-wide design which includes the specification of system hardware.

Acknowledgments

The authors wish to thank the referees for their constructive comments and suggestions about the earlier version of this paper.

References

Adiga, S., 1993, *Object-Oriented Software for Manufacturing Systems* (Chapman & Hall).

Basili, V. R., Briand, L. C., and Melo, W. L., 1996, How reuse influences productivity in object-oriented systems. *Communications of the ACM*, **39**, 104–116.

Booch, G., 1994, *Object-Oriented Analysis and Design with Applications* (The Benjamin/Cummings Publishing Company).

Booch, G., and Rumbaugh, J., 1995, *Unified method for object-oriented development* (document set, Version 0.8, Rational Software Corporation).

Briand, C., and Esteban, P., 1995, An object-oriented and Petri net based approach for real time control of FMSs. *Proceedings of 1995 INRIA/IEEE Symposium on Emerging Technologies and Factory Automation*, vol. 1, pp. 115–123.

Carrie, A., 1988, *Simulation of Manufacturing Systems* (Chichester, UK: Wiley).

Chodosh, D. F, Kamholz, K., Levinson, S. H. and Rhine-smith, R., 1986, Automated chemical synthesis. Part 4: Batch type reactor automation and real-time software design. *Journal of Automatic Chemistry*, **8**, 106–121.

Corkan, A. L., and Lindsey, J. S., 1992, Experiment manager software for an automated chemistry workstation, including a scheduler for parallel experimentation. *Chemonmetrics and Intelligent Laboratory Systems: Laboratory Information Management*, Vol. 17 (Amsterdam: Elsevier Science Publishers B.V.), pp. 47–74.

Dettmer, R., 1995, A class act, the rise of object-oriented technology. *IEE Review*, Nov., 253–256.

Elia, C., and Menga, G., 1994, Object-oriented design of flexible manufacturing systems. *Computer control of Flexible Manufacturing Systems: Research and Development*, S. B. Joshi and J. S. Smith (Eds) (Chapman & Hall), pp. 315–342.

Fujita, M., Usui, S., Kiyama, M., Kambara, H., Murakawa, K., Suzuki, S., Sambe, H. and Takachi, K., 1990, Chemical robot for emzymatic reactions and extraction processes of DNA in DNA sequence analysis. *BioTechniques*, **9**, 584–591.

Hindrey, M. G. and Jarvis, S. A., 1995, *Concurrent Systems: Formal Development in CSP* (McGraw-Hill).

Hoare, C. A. R., 1985, *Communicating Sequential Processes* (UK: Prentice Hall International).

Jackson, D. M., 1989, *The specification of aircraft engine control software using Timed CSP*, Masters thesis, Programming Research Group, University of Oxford, UK.

Jacobson, I., 1996, *Object-Oriented Software Engineering, A Use Case Driven Approach* (Addison Wesley).

Kramer, G. W., and Fuchs, P. L., 1986, Robotics automation in organic synthesis, *Advances in Laboratory Automation Robotics*, vol 3 (Zymark & Hopkinton, MA), pp. 361–372.

Lau, H., 1990. The design of safety critical software using CSP. *Digest of IEE Colloquium in Safety Critical Software for Vehicle and Traffic Control*, 1990/031 8/1–5.

Liscouski, J., 1995, *Laboratory and Scientific Computing: A Strategic Approach* (John Wiley & Sons).

Paulk, M. C., 1995, The evolution of the SEI's Capability Maturity Model for software. *Software Process*, Pilot Issue, 3–17.

PMA'S Computer System Validation Committee, 1986, Validation concepts for computer systems used in the manufacture of drug products. *Pharmaceutical Technology*, **10**, 24–34.

Popkin Software & Systems Ltd., 1995, *CASE Tools for the Real World*, **1**, (1).

Select Software Tools plc., 1996, *The SELECT Perspective, Extending Rumbaugh's OMT for Client/Server Systems Development* (Select Software Tools plc, UK).

Stamper, R., 1990, *The specification of AGV control software using Timed CSP*. Masters thesis, Programming Research Group, University of Oxford, UK.

Yourdon, E., 1989, *Modern Structured Analysis* (Englewood Cliffs, N.J.: Yourdon Press).