Using Problem Frames and projections to analyze requirements for distributed systems

Conference or Workshop Item

Using Problem Frames and Projections to Analyze Requirements for Distributed Systems

Charles B. Haley, Robin C. Laney, Bashar Nuseibeh

Department of Computing, The Open University,
Walton Hall, Milton Keynes MK7 6AA, UK
{C.B.Haley, R.C.Laney, B.Nuseibeh} @ open.ac.uk

Abstract. Subproblems in a problem frames decomposition frequently make use of projections of the complete problem context. One specific use of projections occurs when an eventual implementation will be distributed, in which case a subproblem must interact with (*use*) the machine in a projection that represents another subproblem. We refer to subproblems used in this way as *services*, and propose an extension to projections to represent services as a special connection domain between subproblems. The extension provides significant benefits: verification of the symmetry of the interfaces, exposure of the machine-to-machine interactions, and prevention of accidental introduction of shared state. The extension's usefulness is validated using a case study.

1 Introduction

Architectural considerations often play a part during requirements analysis [4, 9]. For example, reliability, safety, and performance requirements can push towards or away from using a distributed architecture, which will most likely have a profound impact on the specifications derived from the requirements and can provoke changes in the requirements themselves. For example, a requirement to separately deliver components found in a problem can give rise to a form of distributed implementation, e.g. if the traffic light unit described in [5] was designed along with the lights controller but also sold as a separate product. Even if the requirements do not 'force' a distributed architecture, one might wish to analyze the requirements of the system as if it would be distributed, as an aid to predicting architectural consequences of the choices made.

This paper discusses how one might use *problem frames* [5] to structure and analyze problems that for whatever reason might have a distributed *systems architecture*, as opposed to a distributed *software architecture*. Problem frames analysis is about the problem as seen from the world. The problem (the requirement) is stated in terms of measurable and visible effects the system is to have on the world, not in terms of objects and classes visible within the software.

The fact that a problem frames analysis always includes the real (physical) domains suggests that the method could better support analysis of a distributed architecture's influence on requirements, compared to other methods such as KAOS [6] or *i** [8] that do not naturally model the physical domains. In particular, problem frames

analysis always includes a *machine*, representing the computer that will run the software that does the necessary transformations to solve the problem. On the other hand, it is possible that the machine might represent multiple computers, which would mask the distributed nature of the architecture. Finally, a problem frames analysis could create implicit dependencies on state shared between domains (e.g. between computers), something that should not be permitted in a distributed architecture. This paper proposes an extension to problem frames to resolve these difficulties.

The remainder of the paper is structured as follows. Section 2 presents an overview of problem frames. Section 3 elaborates upon the difficulties briefly presented above, and describes the proposed extension. Section 4 presents a case study to validate the extension: a lighting control system which is an expanded version of the example in [2][1]. Section 5 discusses the lessons learned from the case study, and section 6 concludes.

2 Problem Frames

2.1 Problems & Domains

When using problem frames, problems are analyzed by describing the interaction of *domains* that exist in the *world*. The problem frames notation captures domains in a problem along with the interconnections between them. For example, assume that the requirements elicitation process for an automatic door produces the requirement *when a door-open button is pushed, the door shall be opened for 30 seconds*. The requirement states the problem – what is expected to happen in the world, when.

Figure 1 illustrates one set of domains that could satisfy the requirement: a basic automatic door system with three domains, two of which are *given* and one of which is *designed*. One given domain is the door mechanism domain, capable of opening and shutting the door. The second given domain is the one requesting that the door be opened; this domain includes both the 'button' to be pushed and the human pushing the button. The designed domain is the *machine*, the domain that will bridge the gap between the other two domains in order to fulfill the requirement that the door open when the button is pushed. The oval presents the requirement to be satisfied. The text in the 'folded paper' boxes presents the *frame argument*, arguing how and why the requirement is satisfied.

Every domain has *interfaces*, which are defined by the *phenomena* visible to other domains. Phenomena (e.g. events and signals) are visible: they can be observed. The notation shows the phenomena shared between two domains by labeling the line between the domains, then using that label in a box listing the phenomena. Phenomena are listed by indicating the domain controlling the phenomena (the letters before the '!') followed by a list of phenomena within '{' … '}' characters. In Figure 1, we see that the Person + Button domain (PB) controls the event phenomena buttonDown

[1] Although developed independently, the scenario resembles one found in [10]. The major differences are multiple control interfaces, incorporation of security requirements, and dynamic definition of 'rooms' for control purposes.

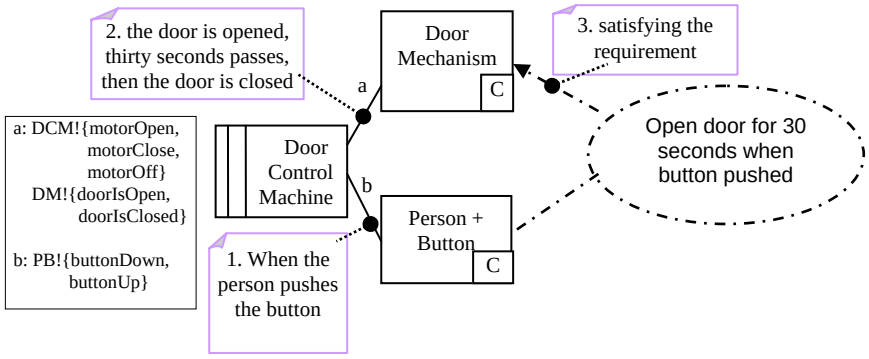


Figure 1. Basic Problem Frames Diagram

and buttonUp. The Door Control Machine (DCM) controls the Boolean phenomena motorOpen and motorClose (turn on and off the motor, set its direction) on the interface between the machine and the Door Mechanism (DM). DM controls the Boolean phenomena doorIsOpen and doorIsClosed.

Requirements are *optative*, describing *desired behavior* instead of existing behavior [5]. Descriptions of the *actual behavior* of given domains (their phenomena: inputs, outputs, and states visible at their interfaces) are *indicative*; they describe an "objective truth" about the behavior of the domain. Indicative domain properties are normally expected to be constant, e.g. the same stimulus in the same context produces the same response. Consider the pushbutton in the domain shown in Figure 1; when the button is pushed, the circuit connected to the button is closed. Putting aside safety and security concerns, we can say that regardless of the state of the system, pushing the button will cause the phenomena to appear on the interface.

Descriptions of the desired behavior of designed domains are optative. As the machine is considered a designed domain, the descriptions of phenomena controlled by it are optative. They describe characteristics that the requirements engineer desires to be true. The job of the software engineers is to produce software that converts these descriptions from optative to indicative. When all phenomena in a system are indicative (and again putting aside many concerns such as safety, security, initialization, and the like), the system is complete.

2.2 Requirements and Specifications

According to Zave and Jackson [11], a *requirement* is an optative description of what the system is to do. Requirements describe a *desired effect*, or a *goal*. Jackson [5] describes a requirement as "the effects in the problem domain that […] the machine is to guarantee." KAOS [6] defines requirements in terms of *agents* : a goal is "an objective the system under consideration should achieve", and a requirement is a goal that can be achieved by a single *software agent* [7]. The *i** framework definition that does not go quite as far: goals model the intentions of stakeholders [8].

Again referring to Zave & Jackson, *specifications* are about phenomena. The specification of a domain is a description of its behavior in terms of the phenomena,

indicative and optative, visible at its interface. The specification of a system is the collection of domain specifications that together fulfill the requirement(s).

The distinction between requirement and specification is an important one. The requirement described the effect desired in the world. The specification describes the interplay of phenomena that will achieve the desired effect.

2.3 Problems, Subproblems, the Context, and Projections

All but the most trivial problems will have multiple requirements. Using problem frames, the analyst proceeds by separating, decomposing, and composing requirements until the individual requirements each fit within a problem class. Each requirement is described using an appropriate problem frame class; these are called *subproblems*. A *context diagram* summarizes all the subproblem diagrams, showing the domains in the union of the subproblems and the interfaces the domains share.

Each subproblem is a *projection* of the context. All domains in the context required to describe a subproblem must appear in the projection; the domains in the context projection represent the world as seen by that subproblem. In some cases multiple domains in the context are projected as a single domain in a subproblem. Domains that are designed in one subproblem appear as given domains in another.

Projections of the context (discussed at length in [5] and briefly but more formally in [3]) are very similar to projections in relational databases [1]. A projection of a relational database table is a new table containing a (potentially improper) subset of columns, and a projection of a problem context is a new context containing a subset of the domains in the problem. The context of a subproblem is a projection of the context of the problem, limiting the domains and/or phenomena in the subproblem to those needed to describe the subproblem.

3 Problem Frames & Distributed Architectures

In problem frames, one effect of choosing a distributed architecture is that *machines* in different subproblems (projections) may in fact represent different physical machines. These machines can communicate with each other (share an interface), resulting in visible shared phenomena. Although they might appear as the machine domain in the context, they must be treated as separate domains for analysis purposes.

To detect this situation, one must identify the potential units of distribution. Each unit of distribution will be represented by at least one problem frame diagram (a projection of the context) showing the machine for that unit of distribution as a machine domain. Other units of distribution that participate in the analysis appear as causal domains in this projection. We say that the subproblem being designed is *using* the unit of distribution being projected as a causal domain. The subproblem (SP) being designed is called the *userSP* and the subproblem being used is called the *usedSP*.

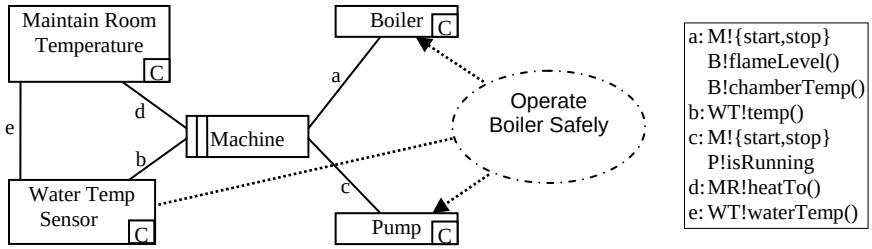
Figure 2 presents an example of a small distributed system, a heating control system similar to the one described in [5]. It measures inside and outside air & water temperatures to anticipate the correct water temperature required to maintain the room

at the desired temperature. There are two subproblems, one representing the boiler controller and the other the heat control function of the room thermostat(s). The thermostat subproblem Maintain Room Temperature (the userSP), uses a projection to represent the furnace controller subproblem Operate Boiler Safely (the usedSP) to supply water at the needed temperature. Maintain Room Temperature does not care how the furnace is controlled. It wants heated water, and controls the heatTo(temp) phenomenon on its interface with Operate Boiler Safely to accomplish that goal.

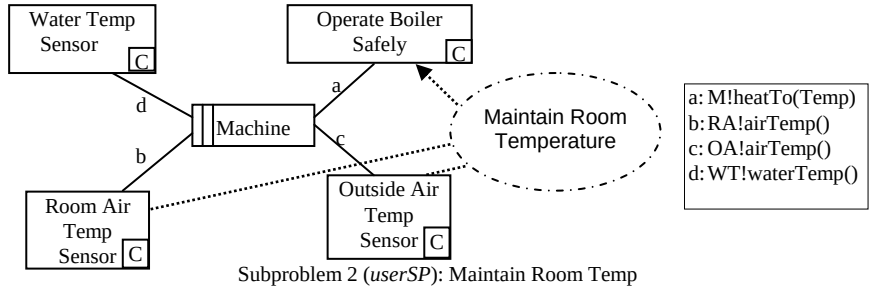
Similarly, Operate Boiler Safely contains a projection of Maintain Room Temperature, representing the thermostats problem. One must insert the projection of Maintain Room Temperature into the subproblem in order to show what triggers the boiler to supply hot water; the thermostat is part of the boiler's world. The boiler does not care why it is delivering hot water or how the decision is made to ask for hot water. It merely supplies hot water when asked.

The interactions in the example illustrate the specific form of decomposition that arises when considering distribution. By carefully tracing the phenomena through the projections, one finds that instead of controlling one of the causal domains in Operate Boiler Safely, Maintain Room Temperature is controlling the subproblem's *machine*. When such a machine-to-machine interface occurs, we say that userSP is using usedSP as a *service*. In the example, Maintain Room Temp is using Operate Boiler Safely as a service to supply heated water.

Several problems are exposed in the above discussion. One is related to symmetry: each subproblem contains a projection of the other, but there is nothing that indicates that the projections are symmetric, or even if they should be. There is nothing in either diagram that directly exposes the machine-to-machine nature of the interfaces, hiding information that is important when considering particular concerns such as



Subproblem 1 (*usedSP*): Operate Boiler Safely



Subproblem 2 (*userSP*): Maintain Room Temp

Figure 2. Heat control system as subproblems

interference, concurrency, and initialization [5]. Causal domains, (e.g. 'Water Temp sensor'), appear in multiple subproblems, potentially introducing shared state and thereby preventing distribution in the recomposed solution.

These difficulties can be resolved by inserting a *connection pseudo-domain* into both projections, making the connections between userSP and usedSP explicit and symmetric. The inserted domain is a *pseudo-domain* because it is fictitious, not representing something physical in the problem. It is a *connection domain* because it represents the point through which users of a service connect to the subproblem supplying the service. When inserted into a userSP, the pseudo-domain represents the projection of the domains a subproblem supplying a service intends to make visible (the machine and possibly some other domains). When inserted into a usedSP, the pseudo-domain represents the projection of the subproblems requesting the service. We give these connection pseudo-domains the name *projection domains*.

To better support validation of symmetry, we propose a strict definition/reference relationship between the one subproblem that *defines* the service and subproblem(s) that *use* the service. A *defining occurrence* is a projection domain in the subproblem that *provides* the service (the usedSP, Operate Boiler Safely in Figure 2). Within the usedSP, the defining occurrence represents all the subproblems that use the service. It acts as a causal domain within the subproblem. The phenomena on its interfaces are the phenomena made available by the service to the userSPs and phenomena that the service expects the userSPs to control.

When a subproblem *uses* the service, the subproblem contains a *using occurrence* projection domain. The using occurrence acts as a causal domain within the using subproblem. It has the same phenomena on its interfaces as the defining occurrence.

There are two properties that must be preserved between a using occurrence and its defining occurrence. The first is completeness: all phenomena appearing on an interface of the using occurrence must appear on an interface of the defining occurrence (or perhaps said to be *optional*, a possibility not further discussed here), and vice versa. The second is directionality of control of phenomena: all phenomena controlled by the using occurrence must be controlled by a domain on one of the defining occurrence's interfaces, and all phenomena controlled by the defining occurrence must be controlled by some domain sharing an interface with the using occurrence.

A defining occurrence is indicated on the problem frame diagram by a projection domain with type **D** (**D**efining). The defining and using occurrences are connected by name; the name of the defining occurrence must be unique across the set of subproblems. A using occurrence is indicated by a projection domain with type **U** (**U**sing).

Figure 3 presents the heating control example from Figure 2 again, this time using projection domains. A defining occurrence is added to subproblem one, Operate Boiler Safely. This defining occurrence, *Operate Boiler*, controls the heatTo phenomenon on the interface between it and the machine. The waterTemp phenomenon is on the interface between the defining occurrence and the Water Temp domain, controlled by Water Temp. Subproblem two, Maintain Room Temp, contains a using occurrence standing for the boiler operation service. The using occurrence is connected by name to the defining occurrence. The using occurrence has the same phenomena on its interface as the defining occurrence, preserving completeness. The

using occurrence controls the waterTemp phenomenon and the defining occurrence controls the heatTo phenomenon, preserving directionality.

The use of projection domains satisfactorily resolves the difficulties listed this section. All interfaces are completely symmetric; *all* the phenomena that a using occurrence can use must be found on an interface on the defining occurrence and vice versa, and it is possible to verify this symmetry automatically. Phenomena that might lead to shared state problems pass through the projection domains. The defining occurrence exposes the machine-to-machine nature of the communication and indicates that the subproblem is to be considered a unit of distribution.

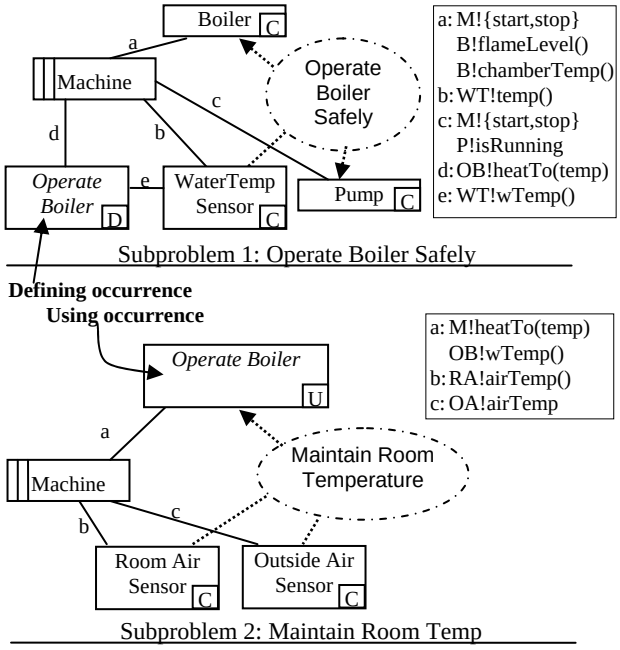


Figure 3. Heat system with projection domains

4 The Lighting Control System Case Study

The lighting control system must conform to the following rough problem statement:

- The system consists of switches and lighting units (lights) associated with a room. When a switch is actuated, the lights in the room must be turned on or off.
- Switches are up/down momentary contact: up turns the lights on and down turns the lights off.
- A master control panel must be included, indicating the state of the lighting units in each room. The indicator on the panel shows green if lights are one, otherwise the indicator does not glow. The state of the lights can be changed using the panel.
- The control panel and lights in 'secure rooms' are to be usable only by people with an appropriate level of authorization. Users carry an identity card (a proximity badge) that is read by a proximity reader either embedded in or installed next to a switch. Lack of a card means the person has the lowest level of authorization possible. The system must record who operated the lights in a secured room. A person who lacks authorization may not change the state of the lights.
- All light on and off actions must be printed on a printer in the control room. If this printer is not working correctly, an alarm of some kind must be given.

– The system must monitor the lighting units. If a lighting unit is not in the correct state (e.g. off when it should be on, or not responding at all), the system must try to correct it. If the correction fails, the system must indicate this fact by changing the indicator on the master control panel of the room containing the failing lighting unit to show red and logging on the printer discussed above.

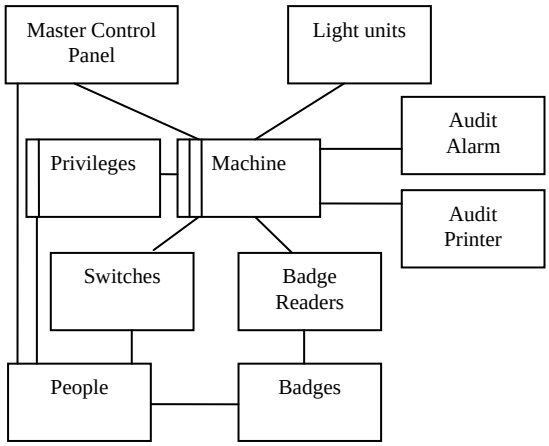


Figure 4. The context diagram

– Failure of any single component in the system shall not affect more than one floor of the building.

4.1 The Light Control Context Diagram

The context diagram must take into consideration several important parts of the problem that the problem statement does not make explicit. For example, the relationship between people and badges must be made clear. The badge identifies the person to the system, and establishes the person's privileges. The privileges determine whether the switch actuation is to be honored. Therefore, the person, the badge, and the privileges are important parts of the problem and should be included in the context diagram. After doing so, we have the diagram shown in Figure 4.

The problem statement contains a requirement stipulating that the eventual implementation must be fault tolerant. Two choices are available: redundancy and distribution. This analysis will explicitly accommodate distribution.

4.2 Subproblem Diagrams

4.2.1 Initial Thoughts. There is nothing physical that relates a switch to the lights it controls or to the logical room that contains the lights. Equally, there is nothing physical that relates a badge reader to a switch or to a room, or relates a badge to a person. It seems that the notion of *room* is a unifying concept fundamental to the problem, and perhaps the problem should be decomposed along that dimension.

Actuating a switch is a request that the state of the lights in a room be changed. From the user's point of view (and the switch's as well), the lights in a room are treated as a unit. It makes sense, therefore, to incorporate the notion of *room* into the switch phenomena along with the *up* and *down* phenomena. A method to map switches and lights to rooms is required. Following this line of reasoning further, it becomes clear that the badge and privilege determination are separate from the switch

actuation. A badge is associated with a person and privilege is associated with a person/room pair, meaning we need another map. We thus end up with the lexical domains *People* → *Privileges*, *Switches* → *Rooms*, and *Rooms* → *Lights*, where the symbol → is read as *maps to*.

One of the fundamental problems, controlling the lights, seems to be a *commanded behavior* problem. People are commanding the lights using the switches and the master panel. However, it appears that the master panel presents enough differences from use of the 'normal' switches to justify separating the two into distinct sets of subproblems, Switches & Lights and Master Control Panel.

We must next consider the Audit problem, which responds to the parts of the problem statement requiring verification that the lights are in the state that they should be. The last problem is the maintenance of the lexical domains.

Please note: to simplify diagrams, most phenomena are not shown in the subproblem diagrams. Also, frame arguments run clockwise from the requirement oval.

4.2.2 Switches & Lights Problem. Accepting this first analysis, we start by connecting the switches to the lights in the rooms that the switches control. This is a commanded behavior problem. The requirement, derived from the system requirements and roughly stated, is *if the user actuates a switch, then the lights in the room(s) associated with the toggle be put into the state indicated by whether the toggle was lifted or lowered.* Clearly we need to associate both switches with rooms and lights with rooms. Two lexical domains will be used for this purpose; for this paper we assume that they will be implemented using a reliable distributed database and do not further consider their reliability and distribution properties. We also decide to separate interpreting the switches from controlling the lights to provide appropriate units of distribution to meet the reliability requirement. The solution would seem straightforward, except that we must account for security.

The security requirement is, again roughly stated, *if a room is secured, then only people with the appropriate permission can cause a state change in the lights*. People are identified by badges. Unfortunately, badges do not indicate who is in a room, but instead indicate who is near a reader. We can reduce the complexity of determining who is 'in' a room by introducing a model that uses badge reader events to maintain a database of who is 'in' a room. This model will be, in effect, the interface between the lights control problem and the badge reader problem. *Enter* and *exit* events generated by the badge reader give the information needed to build the model. A person is considered 'in' a room and able to control a room between enter and exit events. The model is used by other subproblems that

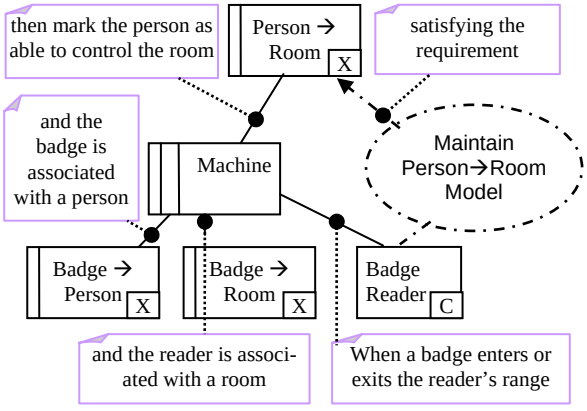


Figure 5. Building the person → room model

verify permissions and enforce security.

Following this route, we find we have two subproblems, one to build the Person → Room model and one (or more) to use it. Figure 5 presents the first subproblem – constructing the model.

We now turn to the subproblems to control the lights. The first subproblem, named Honor Switches and shown in Figure 6, watches for switch events, determines which room is being controlled, and then passes appropriate events to a subproblem that verifies security. To ensure that the subproblems are separable and distributable, the controlled domain is a service, indicated by the using occurrence named *Enforce Security*, (see below). The phenomena passed to *Enforce Security* are shown on the diagrams; they are **on(room)** and **off(room)**. Note: Figure 6 show a notational convenience used throughout this paper: names of projection domains are shown in *italics* as well as by their definition-type letter (D or R).

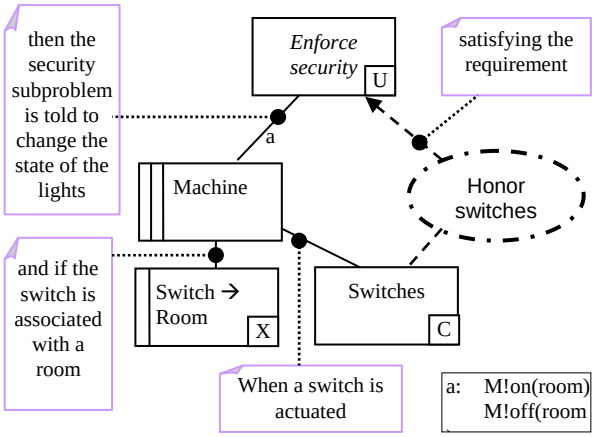


Figure 6. Honor switches – lights control with security

Figure 7 shows the required behavior problem Enforce Security that Honor Switches uses as a service. Enforce Security accepts the **on** and **off** phenomena produced by Honor Switches, then checks to see if the room is secure. If the room is secured (it is in the *Person → Room* model, perhaps with no people in it) then verifies that at least one person near a panel for the room is permitted to control the lights for that room. If permitted or if the room is not secured, it passes the events along through the using occurrence *Units in room*, defined in Control Units in Room (Figure 8). The phenomena passed along are of the form **on(room, person)** and **off(room, person).**

We end with the diagram in Figure 8, Control Units in Room where the defining occurrence *Units in room* is found. This is a commanded behavior problem, looking up which lights are associated with the room and controlling them appropriately. It informs the Maintain MP Indicators subproblem (discussed in the next section) what it did using the service

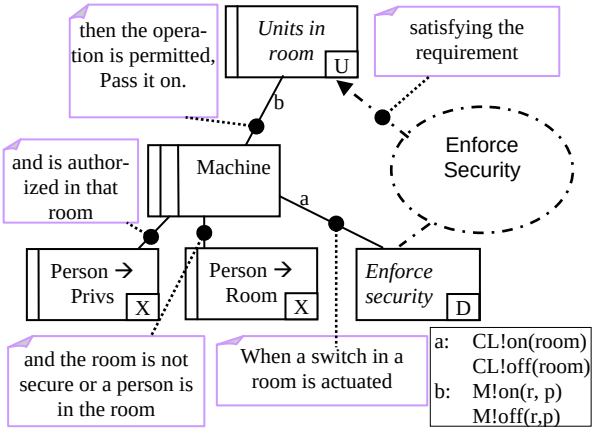


Figure 7. Enforce security

indicated by the using occurrence *Set MP indicator*.

4.2.3 The Master Control Panel. The Master Control Panel problem is decomposed into three subproblems. The first, shown in Figure 9, is an information display problem in which the indicators are set appropriately and the audit trail is maintained. It contains the defining occurrence *Set MP indicator* through which it accepts on and off phenomena from the Control Units in Room subproblem.

The second subproblem concerns controlling the lighting units using the master panel. Shown in Figure 10, it is a commanded behavior problem where pushing a button associated with a room inverts the state of the lights in that room. It uses the service represented by the using occurrence *Units in room* (see Figure 8) to control the lights.

The third subproblem is concerned with master panel security, and is a required behavior problem. As this subproblem is almost identical to the Enforce Security problem presented in Figure 6, the subproblem will not be further discussed.

4.2.4 The Audit Subproblems. The Audit problem is decomposed into two information display subproblems and one commanded behav-

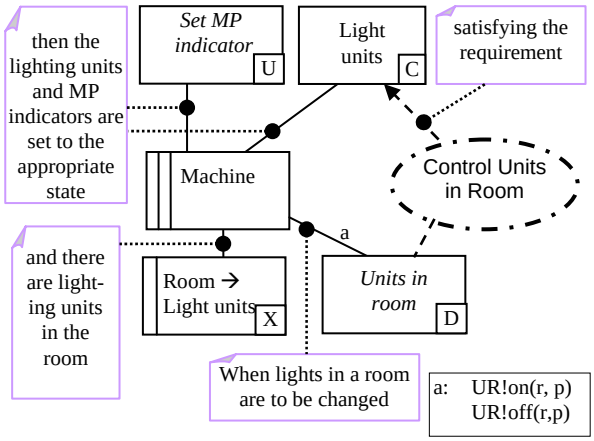


Figure 9. Control units in room

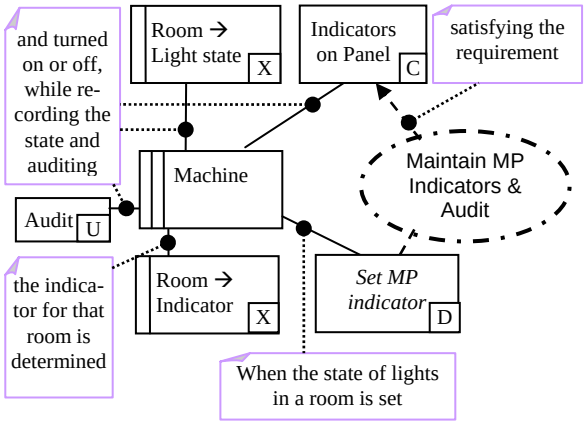


Figure 8. Master control panel

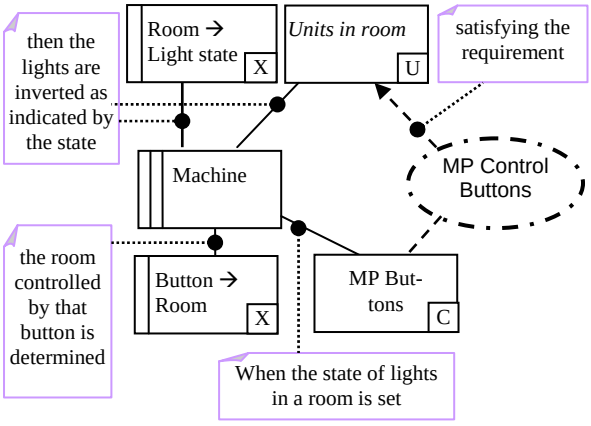


Figure 10. Master control panel buttons

ior subproblem. The first information display subproblem, Audit lights unit shown in Figure 11, scans the lights in each room to determine if they are in the proper state. The fault indicator on the MP is lit via the projection domain *MP fault indicator* if a unit is not in the correct state.

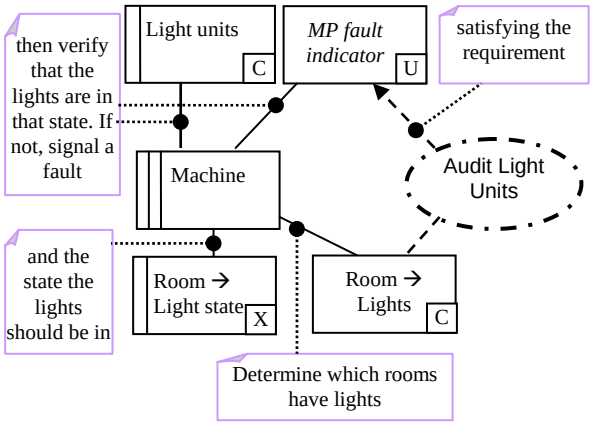


Figure 11. Audit light units

The information display subproblem containing the defining occurrence *MP fault indicator* is similar to Figure 9, as is the subproblem defining the projection domain *Audit*. These subproblems are not further discussed.

The job of the commanded behavior problem is to put the lights into the state they should be in. It is identical to the information display problem in Figure 11, except that it would use the service represented using occurrence *Units in room*, defined in Figure 8.

4.2.5 The Lexical Domains. Several lexical domains have been used in the above diagrams. The creation and maintenance of each of these is described by a simple workpieces problem frame. The subproblems are all very similar and have solutions well described in [5], and they won't be further discussed.

5 Discussion

There are several issues that were not resolved by the use of projection domains in by case study. Some of these derive from the *particular concerns* discussed in [5].

5.1 Distribution

This paper argues that projection domains help with ensuring that a system can be distributed, and the case study supports this assertion. There are, however, some cases where projection domains are not sufficient. For example, the existence of hidden shared state could force merging. A similar question must be asked about lexical domains to determine if they can be used in a distributed fashion (for example as a distributed database).

As problem frames phenomena are considered 'shared', one could argue that distribution is *never* allowed because it breaks the simultaneity assumptions of problem frames analysis. Ignored connection domains create similar difficulties. For example, guards to be evaluated in one subproblem could be added to events in another sub-

problem, creating an implicit connection domain – the guard itself. Use of projection domains does not facilitate or prevent such uses of guards.

It would be very nice to have a better understanding of, and a way to specify, the cases that force merging of the machines. Indicating the simultaneity and concurrency assumptions at an interface would help enormously.

Projection domains assist with determining whether distribution is acceptable by specifying the interface between a defining occurrence and its using occurrences. Indicating the cardinality at these interfaces as described in [2] would provide more information, as cardinalities other than 1:1 imply that some support for concurrency and distribution was intended by the analyst.

5.2 Concurrency

Concurrency problems exist on at least two levels. The first is rather large, exemplified by lexical domains and models. There is an inherent concurrency problem between a machine that maintains a lexical domain and a machine that uses it. The problem manifests itself as inconsistent or partial state. It would seem that this sort of problem is amenable to solution, at least at the phenomena level, by applying transaction semantics to the phenomena.

The second level can be illustrated by looking at the example presented in this paper. It is perfectly permissible to have multiple switches for the same room. The switches and lights in a room might not be controlled by the same computer, leading to potential race conditions as the switches are actuated. Clearly the nature and severity of the concurrency problems depend on how the system is distributed.

5.3 Initialization

Projection domains do not directly solve initialization concerns related to distributed systems. Some of these initialization concerns might be:

5.3.1 What about partial power failures, where parts of the system lose power and parts do not? There are several sub-questions that might arise while discussing this point. Does a partial power failure trigger a safety concern? Can power be lost to parts of the control system, and if so what is to occur while power is lost and when power is restored? The problem is complicated by use of a distributed implementation, as different parts of the system could be ‘off’ at any given time.

5.3.2 The audit process cannot run until system is initialized. This is an example of initialization sequencing. The audit system depends on having the various lexical domains correctly initialized and the lights in a known state. The point after which auditing can start must be determined, then a required behavior problem frame must be added to express the requirement.

5.3.3 Lights added to a room may be in an incorrect state. A maintenance engineer may repair or replace a lighting unit while the system is running. Doing so raises concurrency concerns (maintenance of the lexical domains), correctness concerns (the

newly installed light is off when it should be on and vice versa), identities concerns (movement of units from another room), etc.

5.4 Identities

There are many identities concerns. Most of them are recognized by the inclusion of the lexical domains (the → maps). Some, however, cannot be satisfied with the domains. For example, a switch might be added to the system a long time before it is associated with room. Similarly, a lamp might be added long before it is associated with a room. Badge readers present a similar problem, as does maintaining the correspondence between badges and people.

Another identities concern that will provoke changes to the solution comes from the assumption that switches are in rooms and badge readers are in rooms, therefore someone in the room is actuating a switch. This assertion is clearly incorrect if multiple badge reader/switch pairs are associated with a room. We can confuse the identity of a person at the switch with a person at another switch for the same room. The solution is to map both badges and switches to a pair *(room, location)* instead of to *room*. The diagram in Figure 5 would be changed to build a Person at Location model. The diagram in Figure 10 would be changed to use the Person at Location model. Finally, the diagram in Figure 6 would be changed to use a *Switches → Rooms/Location* map.

5.5 Interference

There are interference or concurrency questions that projection domains do not automatically answer. For example, without care the Audit machine can busily undo Honor Switches actions. Interactions between the audit information display and audit setting the correct light state could make panel indicators flash. If two switches control the same room and one switch commands *off* while the other commands *on*, individual lights could be left in conflicting states. Inconsistent states while maintaining the lexical domains is another source of errors.

6 Conclusions

The case study showed that projection domains help with modeling machine to machine interfaces, something that is necessary when a system's implementation is to be distributed. Projection domains helped keep the subproblems focused while specifying how the subproblems interact. They preserved completeness and directionality, providing a way to verify that all phenomena used and controlled by the defining subproblem were controlled and used by the using subproblem(s), and vice versa. They better encapsulated the service, as the phenomena visible at the projection's interface were defined by the defining occurrence and not by the subproblem using the service. They also provided a form of continuous composition by specifying the interface between a defining occurrence and its using occurrence(s).

Although projection domains resolved some problems encountered when modeling distributed systems, the case study showed that more remain. Future work will focus on ensuring consistent use of lexical domains by multiple subproblems, verify-

ing the semantics of shared phenomena and their parameters, and describing and verifying the concurrency properties of domains and subproblems.

The extension proposed in this paper could be helpful during decomposition even when the result will not be distributed. For example, analysts working on different subproblems may wish to formalize how the subproblems are composed, to 'pre-declare' projections, and to reduce the number of domains included a projection by combining them into a single projection domain.

Acknowledgements

The financial support of the Leverhulme Trust is gratefully acknowledged. We also thank Michael Jackson for his highly pertinent comments, criticism, and help. Finally, we thank the anonymous reviewers for their very helpful criticism.

This paper is a substantially revised version of a paper presented at the 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF'04). Copyright of this earlier version is retained by the authors.

References

[1] Connolly, T., Begg, C., Strachan, A.: Database Systems: A Practical Approach to Design, Implementation, and Management. Second ed.: Addison-Wesley, 1998.

[2] Haley, C.B.: Using Problem Frames with Distributed Architectures: A Case for Cardinality on Interfaces. In *The Second International Software Requirements to Architectures Workshop (STRAW'03) at the International Conference on Software Engineering (ICSE '03)*, Portland OR USA, 9 May 2003.

[3] Hall, J.G., Rapanotti, L.: Towards a Semantics of Problem Frames. Technical Report 2003/05, Department of Computing, The Open University, Milton Keynes UK, 2003.

[4] Hall, J.G., Rapanotti, L.: Problem Frames for Socio-Technical Systems. In *Requirements Engineering for Socio-Technical Systems*, J. L. Maté and A. Silva, Eds., Hershey PA USA: Idea Group Inc., 2004.

[5] Jackson, M.: Problem Frames. Addison Wesley, 2001.

[6] van Lamsweerde, A.: Goal-oriented Requirements Engineering: A Guided Tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, Toronto, Canada: IEEE Computer Society Press, 27-31 Aug 2001, pp. 249-263.

[7] van Lamsweerde, A.: Elaborating Security Requirements by Construction of Intentional Anti-Models. In *Proceedings of ICSE'04*, Edinburgh Scotland, 26-28 May 2004.

[8] Liu, L., Yu, E., Mylopoulos, J.: Security and Privacy Requirements Analysis Within a Social Setting. In *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'03)*, Monteray Bay, CA USA, 8-12 Sept 2003.

[9] Nuseibeh, B.: Weaving Together Requirements and Architectures. *Computer (IEEE)*, 34(3) (Mar 2001), 115-117.

[10] Queins, S., Zimmermann, G., Becker, M., Kronenburg, M., Peper, C., Merz, R., Schäfer, J.: The Light Control Case Study: Problem Description. *Journal of Universal Computer Science*, 6(7) (Jul 2000), 586-596.

[11] Zave, P., Jackson, M.: Four Dark Corners of Requirements Engineering. *Transactions on Software Engineering and Methodology (ACM)*, 6(1) (Jan 1997), 1-30.