Electronic Communications of the EASST
Volume 45 (2011)

Proceedings of the Fourth International Workshop on Formal Methods for Interactive Systems (FMIS 2011)

Modelling Distributed Cognition Systems in PVS

Paolo Masci, Paul Curzon, Ann Blandford, Dominic Furniss

16 pages

Guest Editors: Judy Bowen, Steve Reeves
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Modelling Distributed Cognition Systems in PVS

Paolo Masci[1]*, **Paul Curzon**[1], **Ann Blandford**[2], **Dominic Furniss**[2]

[1] Queen Mary University of London
School of Electronic Engineering and Computer Science

[2] UCLIC, UCL Interaction Centre
University College, London

Abstract:

We report on our efforts to formalise DiCoT, an informal structured approach for analysing complex work systems, such as hospital and day care units, as distributed cognition systems. We focus on DiCoT's information flow model, which describes how information is transformed and propagated in the system. Our contribution is a set of generic models for the specification and verification system PVS. The developed models can be directly mapped to the informal descriptions adopted by human-computer interactions experts. The models can be verified against properties of interest in the PVS theorem prover. Also, the same models can be simulated, thus facilitating analysts to engage with stakeholders when checking the correctness of the model. We trial our ideas on a case study based on a real-world medical system.

Keywords: Higher Order Logic, PVS, Distributed Cognition, DiCoT, Interactive System Design.

1 Introduction and Motivation

Designing a system involves specifying the characteristics that are necessary for it to accomplish given tasks in given environments. Design errors are well-known sources of system failures: in computer systems, they represent the major cause of failures; in interactive systems, they are also a major cause of systematic human errors. For instance, in the healthcare domain the 'system' of importance is often not just a single computer device but the whole work environment. Whether it was explicitly designed, or more likely evolved and was adapted by those working within it over time, errors in its 'design' can cause either kind of failure. The development process of a system of whatever level, therefore, must adopt appropriate means to eliminate design errors. This is specially relevant in safety-critical domains, such as healthcare.

Formal methods represent appealing techniques for eliminating design errors, because they are based on mathematical logic and provide a non-ambiguous language for describing the system. Furthermore, formal methods are supported by analytical tools and methodologies that enable the prediction of possible system behaviours. To date, two main classes of formal approaches have been explored for reasoning about interactive system design: *machine-centred* approaches, which aim to verify functional properties of interactive systems' specifications, such as deadlock-freedom and consistency of assumptions (e.g., [CH97]); and *user-centred* approaches, which

* Corresponding author.

explicitly model human behaviour, and reason about systems where users interact with devices according to a cognitively plausible behaviour (e.g., [RBCB09]). Such formal approaches generally explain information processing in the use of devices at the level of the individual, and point attention at localised phenomena happening between a user and an interactive device. Although useful, this way of reasoning is sometimes insufficient to capture the distributed nature of the mechanisms that usually enable users to carry out their tasks in the real world [RE94]. In fact, humans deliberately use and organise the whole environment to support their behaviour [Nor02]. Hence, when analysing and designing complex interactive systems, experts should consider the whole work system [Thi08], and study collections of humans, devices, artefacts, and their relations to each other in the work practice.

Distributed cognition [FH91] represents a promising conceptual framework for studying the design of complex interactive systems. The idea of distributed cognition is that cognition is not confined in the mind of humans, but it spans across humans and artefacts. As such, cognition is a property of the whole system, and can be described in terms of transformations of the representational state of information. For instance, in familiar everyday interactions we use to-do lists to organise attention to tasks, we use shopping lists to extend our memory and we group piles of paper to organise our work. Kirsh and Maglio [KM94] argue that people will change their external environment to help them with the problem solving space they are working within, i.e. manipulating the world changes the cognitive space. An example of this is moving tiles about when playing Scrabble so players can group letters together and improve their ability to find longer words. Moving toward more complex systems, Hutchins [Hut95] analysed how a cockpit "remembers its speed" through a combination of different people, in different roles, with different tools and artefacts collectively moving and changing the representation of information.

DiCoT [BF06] is an informal structured methodology that has been proposed in the human-computer interaction community for applying the framework of distributed cognition to the design of teamwork settings. The approach has been successfully used to analyse different real-world systems, see, for instance [SRSF06], [MD08] and [FB06]. Briefly, DiCoT proposes five interdependent models to analyse socio-technical systems: the information flow model, physical model, artefact model, social model and evolutionary model. Associated with each of these models is a representation or diagram and distributed cognition principles that are distilled from the literature, e.g. in the physical model the analyst might look for factors influencing the situation awareness in the system, or lack there of. These models are described further below.

DiCoT's models are specified using an informal notation based on a mixture of semi-formal diagrams and natural language. Designers and analysts also reason about properties of the models with informal deduction methods. Whilst the flexibility of informal notation can be useful in some ways, it may lead to ambiguous specifications, which may reduce the repeatability of the analysis. Similarly, the informal deduction methods used in DiCoT are high-level proof sketches, and there is a concrete risk of using hidden assumptions when proving properties.

Contribution. The aim of this work is to support human-computer interaction experts in describing how tasks are carried out in complex work systems, and in identifying error-prone system designs. To this end, we report on our efforts to formalise DiCoT in the specification and verification system PVS [ORR+96]. We focus here on DiCoT's information flow model, which studies how information is transformed and propagated in the system. Our contribution is a set of generic PVS theories that can be used by analysts to describe work practice and

work settings. Such generic theories can be naturally mapped to the informal description of DiCoT's information flow models, i.e., the relationship between the developed theories and the concepts used in DiCoT's information flow models can be easily seen and justified. Also, the developed theories can be executed within the PVS system, thus facilitating analysts to engage with stakeholders when checking the correctness of the model. To trial these ideas, we have applied them to the domain of medical work, and we have formally modelled and analysed a case study based on the London Ambulance Service. The case study is based on already published work presented by [BF06]. With our models and analysis, we were able to spot some potential issues that were not reported in the published study. The original analyst confirmed that having such issues highlighted while the study was carried out would have been useful in gaining a more complete and insightful analysis.

The rest of the paper is organised as follows. In Section 2, we introduce DiCoT's models and present details of the information flow model. In Section 3, we present a brief overview of the PVS specification and verification system. In Sections 4 we report on the developed PVS theories, and explain how the theories can be used to formalise DiCoT's information flow model. In Section 5, we trial our theories by specifying the DiCoT information flow model of the London Ambulance Service. In Section 6, we discuss related work and draw the conclusions.

2 DiCoT Models

DiCoT uses five models to describe work systems. Each model provides a different perspective on the system: the *physical model* studies the physical layout of the system; the *artefact model* studies how artefacts are designed and used in the system; the *information flow model* studies how information is transformed and propagated in the system; the *social model* studies the roles, skills and knowledge in the system; and the *evolutionary model* studies how the system has changed over time. Here, we focus on the information flow model. Such a model highlights the role of the various actors in the system, the content of information items exchanged among actors, and the sequences of actions carried out by actors for processing and exchanging information items. The model consists of a textual description, which specifies the activities carried out in the system in natural language, and a diagram, which graphically depicts the dependency relations among activities. In the following, we show the information flow model for a case study based on the Central Ambulance Control room of the London Ambulance Service. The case study has been described and analysed in [BF06] and [Fur04]. We formalise this case study in Section 5.

2.1 Information Flow Model Example: the London Ambulance Service

The London Ambulance Service (LAS) is the world's largest emergency healthcare system, caring for more than one and a half million patients every year. The ambulance service is coordinated from a central ambulance control room, which consists of two main areas: call taking and dispatching. Operators in the call taking area receive calls from external callers and filter out relevant information about the incident. Operators in the dispatching area use the information entered in the system by call-takers for deciding which ambulance should be allocated to which incident. There is a dispatching area for each zone of the city (London has seven zones).

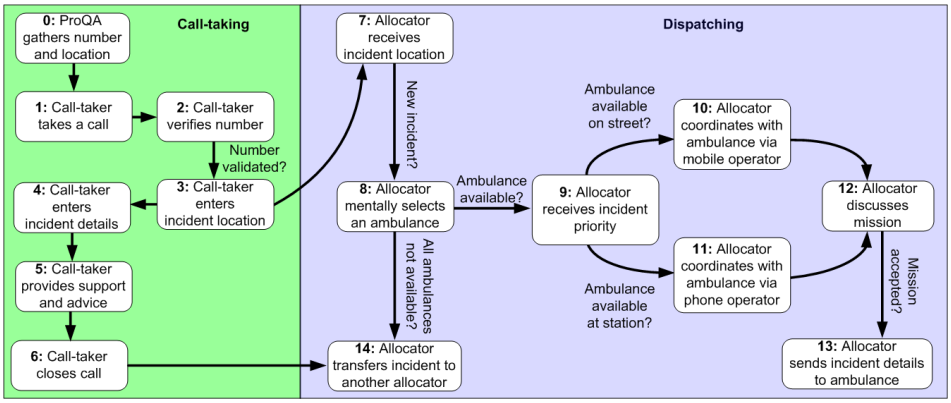

Figure 1: DiCoT Information Flow Model Diagram of the London Ambulance Service.

Textual Description. The informal specification of the activities carried out in the call taking and dispatching areas, based on that from the original analysis, follows. In our description, we will use a level of detail appropriate for the purposes of this article. Readers interested in a more detailed informal specification of the whole system should refer to [Fur04].

Activities carried out in the call taking area. Call-takers interview external callers according to a protocol captured in the 'Advanced Medical Priority Dispatch System'. This protocol defines a structured dialogue between call-takers and external callers that enables call-takers to distill information for classifying incidents in terms of their medical urgency. ProQA [Pri05] is a computerised version of the system, and is currently used in the central ambulance control room. ProQA structures the dialogue between a call-taker and an external caller as follows. First, the call-taker greets the external caller and verifies the caller's location and telephone number, which are automatically entered by ProQA. Second, the call-taker starts a questioning procedure for gathering information on the incident's location and details. This focuses the system's attention on the critical factors needed to prioritise the incident. As soon as the call-taker enters the incident's location in the system, the relevant sector desk is activated to receive live information on the incident. Specifically, the allocator responsible for the incident's zone is notified about the new incident, and is updated in real-time as the call-taker inputs further information. Third, the call-taker provides support and advice, and then closes the call.

Activities carried out in the dispatching area. Allocators can view the position of all ambulances, and they are selectively updated on new incidents when call-takers enter information on them. As soon as a call-taker enters a new incident location, a communication is automatically established between the call-taker and the allocator responsible for the relevant city area, and the allocator is able to view in real-time the incident details entered by the call-taker. While the incident's details are entered, the allocator can start checking that the call is actually a new incident, and not an additional call about an incident that has already been reported. In the case of a new incident, the allocator mentally selects possible ambulances on the basis of their location and availability. The allocator will have good situation awareness about where ambulances are and what incidents they may be attending to. As soon as the incident priority is known, the allocator alerts the ambulance crew and coordinates with it. Depending on the actual position of the ambulance, the coordination between allocator and ambulance crew is supported either by a phone operator (the ambulance is at the station) or a mobile operator (the ambulance is on the

streets). If the ambulance crew accepts the mission, the allocator sends the incident information to the ambulance crew. If no ambulances are available in the allocator's zone, then the allocator transfers the incident's details to another allocator in a neighbouring sector.

Diagram. The diagram was not included in the published work, and we define it here in collaboration with the human-computer interaction experts that conducted the original study. The diagram is shown in Figure 1: labelled boxes represent activities, and edges represent causal and temporal dependency relations among activities. An activity can be performed only if all directly connected activities have already been performed. For instance, activity *"1: Call-taker takes a call"* can be performed only if *"0: ProQA gathers number and location"* has already been performed. Some activities can be performed concurrently, e.g., *"4: Call-taker enters incident location"* and *"7: Allocator receives incident location"*. Edges may have labels that specify control-flow conditions. For instance, the outgoing edges from *"9: Allocator receives incident priority"* have labels *"Ambulance available on street?"* and *"Ambulance available at station?"*, which define two possible ways of continuing the task.

3 Background on PVS

The specification and verification system PVS (Prototype Verification System) [ORR+96] combines an expressive specification language with an interactive proof checker. The PVS specification language builds on classical typed higher-order logic with the usual base types (e.g., `bool`, `nat`, `integer` and `real`), function type constructors `[A -> B]` (predicates are functions with range type `bool`), and abstract data types. The language supports *predicate subtyping*, which is a powerful mechanism to express complex consistency requirements. PVS specifications are packaged as theories. Theories can be parametric in types and constants, and they can use definitions and theorems of other theories by importing them. PVS provides a pre-defined built-in prelude, and a number of loadable libraries that provide a large number of standard definitions and proved facts that can be used when developing new theories. PVS has an automated theorem prover that can be used to interactively apply powerful inference procedures within a sequent calculus framework. The primitive inferences procedures include, among others, propositional and quantifier rules, induction, simplification using decision procedures for equality and linear arithmetic, data and predicate abstraction [ORR+96]. PVS has a ground evaluator [COR+01] that automatically compiles executable constructs of a specification into efficient Lisp code. In order to be able to execute theories that include non-executable constructs (e.g., declarative specifications), the ground evaluator can be augmented by so-called semantic attachments. Through these, the user can supply pieces of Lisp code and attach them to the declarative parts. The ground evaluator was subsequently extended by a component, denominated PVSio [Muñ03], which provides a high-level interface for writing semantic attachments, as well as a set of proof rules to safely integrate the attachments to the theorem prover of PVS.

4 Formal Specification of DiCoT's Information Flow Model

We have developed a customisable set of PVS theories for specifying DiCoT's information flow model. Since one of the key aims of our work is to support human-computer interaction experts

in their analysis, we designed the theories such that a *natural mapping* exists between the PVS specification and DiCoT's information flow model, i.e., the relationship between the developed theories and the concepts used in DiCoT's information flow models can be easily seen and justified. This opens the possibility of enabling human-computer interaction experts to directly use such formal models for specifying the system. The developed theories can be re-used and refined for building models of different systems at different level of details. Also, the models can be animated with PVSio, thus facilitating the dialogue between analysts and stakeholders when checking the correctness of the model. In order to support model animation, we developed a simulation engine that can be conveniently used to automatically schedule activities and generate execution traces. The simulation engine is specified as a higher-order function in PVS, thus the correctness of its specification can be formally proved in the PVS theorem prover.

4.1 Generic Models

In order to build a set of PVS theories that can be naturally mapped to the informal description of DiCoT's information flow model, we draw concepts from Activity Networks [MM84], a widely used formalism for modelling complex concurrent systems, and from approaches for the analysis of protocols for distributed systems of autonomous and cooperating nodes [BMP08, BMP09]. Specifically, we use three basic modelling concepts for developing our PVS theories: system state, activities, and task.

System State. A system state is a snapshot of the value of information items and of the characteristics of system elements. In PVS, system states can be conveniently specified with structured data-types. Each field of the data structure represents either the value of an information item or the characteristics of a system element. We assign unique identifiers to each modelled item; identifiers can be either natural numbers or enumerated types. When using enumerated types, the PVS type system automatically checks that the enumerated identifiers are unique, and automatically generates predicates for recognising the enumerated values (the predicate is given by a function whose name is the value's name followed by a question mark). Since the level of details for specifying a system state generally depends on the property of interest, we exploit *information hiding* when defining theories for the system state, i.e., each PVS theory is assimilated to the class concept used in object-orient programming languages, where interface functions are used for accessing and modifying data types in a consistent way. We will show an example of the system state in Section 5.

Activities. An activity is an action carried out in the system. Activities can be carried out either by humans, devices, or collections of humans and devices, and they are specified as transition functions over system states. We identify each activity with a unique identifier. We developed a PVS theory, `activity_th`, that contains the following definitions: `activity`, a function type suitable for specifying activities as state transitions over system states; `activity_id`, a bounded natural number type for defining unique identifiers for activities; an `execute` function, which specifies that a new system state can be obtained by applying an activity to the current system state. The type definition of system state and the number of activities are theory parameters. We will show examples of specifications in Section 5.

```
activity_th[system_state: TYPE, N_ACTIVITIES: posnat]: THEORY
 BEGIN
```

```
  activity: TYPE = [system_state -> system_state]
  activity_id: TYPE = below(N_ACTIVITIES)
  execute(act: activity): [system_state -> system_state] =
    LAMBDA(sys: system_state): act(sys)
 END activity_th
```

Tasks. A task defines *how* and *when* activities carried out in the system. We specify tasks with a graph-based notation: nodes in the graph represent activities, and edges between nodes represent dependency relations between activities. An activity is enabled when all directly connected activities have already been performed. Whenever an activity becomes enabled, such an activity can be performed. When several activities are enabled at the same time, then such activities can be performed concurrently. Dependency relations among activities can be parametric with respect to control flow conditions, which define different ways of continuing the task. Control flow conditions are specified by associating dependency predicates to edges.

In PVS, we specify tasks as structured data types consisting of four fields: `F`, a function that associates a unique identifier to each activity that can be performed in the task; `S`, a status vector that defines the progress status of each activity in the task; `G`, a directed graph that defines dependency relations among activities (nodes in the graph are identifiers of activities); `P`, a function that associates dependency relations to dependency predicates. The type definition of G uses the NASA library on directed graphs [BS98], which provides a large number of standard definitions and already proved theorems. The type definition of dependency predicates, on the other hand, has been defined as a function from system states to booleans. The type definitions of system state, activity, activity identifier, and activities' progress status are theory parameters, i.e., they are left unspecified, and must be instantiated by the theories that import `task_th`. The theory follows.

```
task_th[system_state, activity, activity_id, progress_status: TYPE]: THEORY
 BEGIN IMPORTING digraphs[activity_id]
   dependency: [system_state -> bool]
   task: TYPE = [# F: [activity_id -> activity],
                   G: digraph[activity_id],
                   P: [edgetype[activity_id] -> dependency],
                   S: [activity_id -> progress_status] #]
END task_th
```

4.2 Simulation Engine

We developed a simulation engine for animating the formal specification. In our work, the main utility of the simulation engine is to facilitate the dialogue between analysts and stakeholders when checking the correctness of the formal specification.

The developed simulation engine takes care of scheduling activities, and uses the PVSio extension for generating a visual feedback of the execution. The engine is defined as a higher-order function, `exec`, that iteratively selects at most `N` times the activity to be performed, and generates the new system state by executing the selected activity. The engine selects the activity on the basis of its progress status (defined in theory `progress_status_th`), which can be one of the following: `ready`, i.e., the activity is ready for execution, `needs_action`, i.e., the activity cannot be performed because other activities need to be completed first, `completed`, i.e.,

the activity has been performed, `cancelled`, i.e., the activity will not be executed (this may happen, e.g., because of control flows); and `deleted`, i.e., the activity has not been executed. Activities are chosen non-deterministically from two work-lists, `C` and `R`. Work-list `C` contains cancelled activities. Work-list `R` contains activities ready for execution. Activities in work-list `C` have priority over those in work-list `R` —this way, dependency relations due to cancelled activities can be automatically removed from the dependency graph of the task. Whenever an activity is selected from a work-list, the activity is also removed from such a work-list. Two auxiliary functions, `update_deleted` and `update_completed`, are used in the engine for updating the status of activities according to (i) the status of the performed activity, and (ii) the dependency relations specified in the task. The execute function defined in `activity_th` is used to generate the new system state when an activity completes (the system state remains unchanged when an activity is deleted). The execution terminates when either `N` steps have been performed or both work-lists `R` and `C` are empty. The theory follows.

```
basic_engine_th[system_state: TYPE, N_ACTIVITIES: posnat,
                state2string: [system_state -> string]]:THEORY
 BEGIN %--imports omitted
  execution_engine: TYPE = [task, system_state -> system_state]
  exec(N: nat): RECURSIVE execution_engine =
    LAMBDA(t: task, sys: system_state):
     IF N = 0 THEN sys
     ELSE LET  dbg = print(state2string(sys)),
               C   = { x: activity_id | cancelled?(S(t)(x))},
               R   = { x: activity_id | ready?(S(t)(x))}
          IN IF empty?(C) AND empty?(R) THEN sys
             ELSE LET (t_prime, sys_prime) =
              COND NOT empty?(C)
                   -> LET x = choose(C)
                      IN (update_deleted(x)(t,sys), sys),
                   NOT empty?(R) AND empty?(C)
                   -> LET x = choose(R)
                      IN (update_completed(x)(t,sys), execute(F(t)(x))(sys))
             ENDCOND IN exec(N-1)(t_prime, sys_prime) ENDIF
     ENDIF MEASURE N
END basic_engine_th
```

5 Case Study

In this section, we use the developed PVS theories for modelling and analysing the DiCoT information flow model of Section 2.1, which describes the activities carried out in the London Ambulance Service. We show how the level of abstraction of the specification can be seamlessly changed by importing different versions of the theories. When importing abstract theories containing only declarations, the specification of the task defines only the overall structure of the task, which is useful for checking a number of consistency constraints related to control flow conditions. When importing concrete theories that provide details on how activities modify the system state, the specification can be used to detect covert dependency relations due to read/write operations on the system state. Also, we show how to use the developed simulation engine.

5.1 High-Level Specification

The high-level specification of the London Ambulance Service is given by the DiCoT diagram reported in Figure 1. The formal specification in PVS was obtained by defining a function, LAS_task, that specifies the dependency relations among activities. The system state and the name of the activities carried out in the task are declared in the imported theory. In the following, we show an excerpt of function LAS_task. In Section 5.1, we show how this high-level specification can be refined in order to detect covert dependency relations. In Section 5.2, we show how the high-level specification can be conveniently used for checking consistency constraints on control flow conditions.

```
LAS_task_th: THEORY
 BEGIN IMPORTING LAS_activities_th,
                 task_th[system_state, activity, activity_id, progress_status],
                 %--more imports omitted
  LAS_task(sys: system_state)
          (task_status: [activity_id -> progress_status]): task =
    LET f: [activity_id -> activity] =
             LAMBDA(id: activity_id):
                COND
              id = 0 -> ProQA_gathers_number_and_location,
              id = 1 -> Call_taker_takes_a_call,
              id = 2 -> Call_taker_verifies_number,
              ...
       g: digraph[activity_id] =
             LAMBDA(id: activity_id):
                 COND
              id = 0 -> {x: activity_id | x=1},
              id = 1 -> {x: activity_id | x=2},
             ...
       p: [edgetype[activity_id] -> dependency] =
             LAMBDA(id1,id2: activity_id):
                 COND
              (id1,id2) = (2,3)   -> Number_validated?,
              (id1,id2) = (7,8)   -> New_incident?,
            ...
    IN (# F := f, G := g, P := p, S := task_status #)
END LAS_task_th
```

5.2 Refining the High-Level Specification

The high-level specification can be refined by specifying the informative content of the system state, and how activities actually modify the system state. Specifically, without changing the specification of function LAS_task, we can add details by importing a different version of theory LAS_activities_th. In the following, we show how we refined the theory.

System State. In order to refine the system state, we need (i) to define the characteristics of system elements, and (ii) to define the informative content of information items.

System Elements. We defined a PVS theory, LAS_actors_th for defining elements' characteristics. Specifically, we defined a new enumerated type, element_id, for identifying actors in the London Ambulance Service, and we used predicate sub-typing for deriving a new type

of identifiers, las_staff, for the staff members of the London Ambulance Service. This new type is useful for specifying activities that can be carried out only by staff members — in PVS, all functions must be total, but partial functions can be specified as total functions over a restricted domain [SO99]. In the theory, we define the characteristics of an ambulance (ambulance_state) in terms of its availability status and location. An excerpt of the theory follows.

```
LAS_actors_th: THEORY
 BEGIN
  element_id: TYPE = {NA, external_caller, call_taker, allocator, ...}
  las_staff : TYPE = {n: element_id | call_taker?(n) OR allocator?(n) OR ...}
  %-- ambulance state
  ambulance_availability: TYPE = {NA, available, not_available, on_duty}
  ambulance_location: TYPE = {NA, on_street, at_station}
  ambulance_state : TYPE = [# status: ambulance_availability,
                              loc   : ambulance_location #]
  %-- more definitions omitted
 END LAS_actors_th
```

Information Items. Information items flowing within the system describe, in this case, information exchanged between call-takers and allocators through the ProQA system. Information items handled by call-takers include incident locations and incident details. Such information items are automatically augmented by the ProQA system, which enters phone numbers of external callers and assigns unique computer aided dispatch (CAD) numbers to calls. Information items handled by allocators include a set of incidents (each of which is uniquely identified by a CAD number), incidents' details, ambulances' location and availability. A PVS theory suitable for modelling the state of such information items contains two structured data types: call_taker_info, which includes information entered by call-takers and information automatically entered by ProQA; and allocator_info, which defines information items handled by allocators. An excerpt of the theory follows.

```
LAS_information_items_th: THEORY
 BEGIN %-- imports omitted
  call_taker_info: TYPE = [# caller_cad       : CAD_number,
                             caller_phone     : phone_number,
                             caller_location  : location,
                             incident_location: street,
                             incident_details : details #]
  allocator_info: TYPE = [# cad_set    : finite_set[CAD_number],
                            incidents  : [CAD_number -> incident_state],
                            ambulances: [ambulance  -> ambulance_state] #]
  %-- more definitions omitted
 END LAS_information_items_th
```

System state. The system state, according to a distributed cognition view, identifies the representational state of information. In our example, we are interested in the representational state of an incident. According to the narrative description given in Section 2.1, we need to consider the items handled by a call taker and an allocator. Hence, the system state can be defined with a

structured data type containing two fields: `call_taker_state`, of type `call_taker_info`, which describes the representational state handled by a call taker, and `allocator_state`, of type `allocator_info`, which describes the representational state handled by an allocator.

```
LAS_system_state_th: THEORY
 BEGIN IMPORTING LAS_information_items_th, LAS_actors_th
  system_state: TYPE = [# call_taker_state: call_taker_info,
                          allocator_state : allocator_info #]
  %-- more definitions omitted
 END LAS_system_state_th
```

Activities. Here we show how to define the first two activities carried out by call-takers with a functional notation that uses the `LET-IN` construct of the PVS syntax. The first activity, *"0: ProQA gathers number and location"*, changes the system state by assigning a unique CAD number and by entering the caller's phone number and location; the second activity, *"1: Call-taker takes a call"* leaves the system state unchanged.

```
LAS_activities_th: THEORY
 BEGIN IMPORTING LAS_system_state_th, activity_th[system_state, N_ACTIVITIES]
  %-- 0: ProQA gathers number and location
  ProQA_gathers_number_and_location: activity =
    LAMBDA(sys: system_state):
     LET cad = new_cad({c: CAD_number | NOT cad_set(allocator_state(sys))(c)}),
         c_phone = ProQA_gathers_phone_number,
         c_location = ProQA_gathers_location,
         c_state = (# caller_cad := cad, caller_phone := c_phone,
                      caller_location := c_location, incident_location := NA,
                      incident_details := NA #)
      IN sys WITH [ call_taker_state := c_state ]
  %-- 1: Call-taker takes a call
  Call_taker_takes_a_call(i: call_taker): activity =
    LAMBDA(sys: system_state): sys
  %-- more definitions omitted
END LAS_activities_th
```

5.3 Formal Analysis

We carried out two kinds of formal analysis on the formal specification of the London Ambulance Service. The first analysis is performed on the high-level specification for ensuring semantic constraints on control-flow conditions. The second analysis aims at comparing the *work practice*, which reports how activities are carried out by users in the real system, against the *reference manual* of the ProQA system, which reports how activities should be carried out in the system according to the system designer's point of view. This second analysis has similarities with that carried out by Rushby in [Rus02]; in his work, Rushby compared the specification of an interactive system with the mental model created by its users for discovering possible sources of mode confusion.

Checking Semantic Constraints. One of the constraints that we need to check in our specifications is that control flow conditions cover all possible situations. We can specify such a

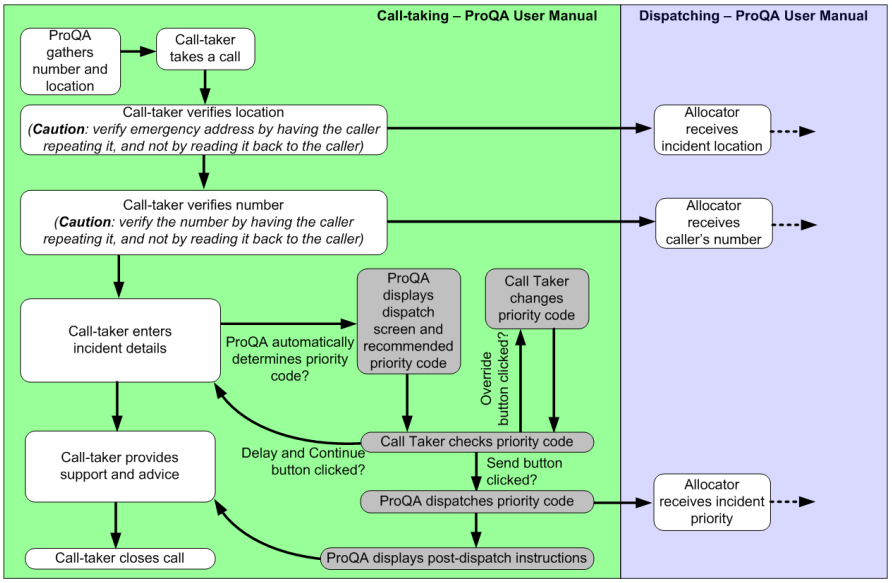

Figure 2: DiCoT's Information Flow Model Diagram of ProQA's Manual (some details of the Dispatching activities are omitted); gray boxes are activities that were not reported in [BF06].

constraint as a predicate on the dependency predicates associated to the incoming and outgoing edges of each node in the graph. We found a violation for the following activities: *"2: call-taker verifies number"*, *"7: allocator receives incident location"*, *"12: allocator discusses mission"*. The issue is probably due to the following hidden assumptions: (i) call-takers are supposed to know what to do when a number cannot be validated and when an ambulance does not accept the mission, and (ii) allocators are supposed to wait for new incidents before performing any action. The utility of this analysis is to force us to expose all the assumptions made on the system and check their validity.

Comparing Work Practice and Reference Manual. The comparison of work practice and reference manual is useful for pointing out possible mismatches between mental models developed by users (i.e., how users understand the functionalities of the system), and the reference specification of the system (i.e., how the system should work according to the designer's point-of-view). In order to trial this possibility, we formalised the ProQA user manual (ver 3.4) [Pri05]. An excerpt of the high-level description of the manual is graphically depicted in Figure 2. In the figure, we use a level of detail appropriate for the purposes of this article.

The exercise of building the specification enabled us to spot several mismatches related to the understanding of *when* and *how* information flows from call-takers to allocators. For instance, according to the description obtained with the ethnographic study performed in [BF06], the mental model developed from speaking to the call-takers seems to describe the flow of information from call-takers to allocators as an automatic procedure guaranteed by the system (in the following, some parts of the text reported in [BF06] are omitted; for such parts we show dots within square brackets): *"As soon as the call-taker enters the incident's location in the system [...] the allocator responsible for the incident's zone is notified about the new incident, and is updated in real-time as the call-taker inputs further information."*. The specification of the user manual, on the other hand, suggests that such communication happens only when the ProQA system dis-

plays a send dispatch screen: *"The send dispatch screen appears as soon as ProQA has enough information to recommend a dispatch code. [...] Click on the Send button to immediately send the dispatch code."* (page 88 of ProQA's user manual [Pri05]). Furthermore, from the manual it emerges that the user may delay sending the information: *"When appropriate, click on the Delay and Continue button to delay dispatch and continue caller interrogation."* (page 89 of ProQA's user manual [Pri05]). This last mismatch is potentially a serious problem, because allocators cannot proceed if call-takers delay sending the dispatch code. The issue, indeed, seems to have been foreseen by the system designers, because ProQA's user manual reports the following warning on delaying the dispatch: *"Exercise caution when delaying dispatch. Do it only when you need to ask additional questions before sending dispatch."* (page 89 of ProQA's user manual [Pri05]). The original analyst confirmed that having such issues highlighted during the study would have been useful in gaining a more complete and insightful analysis.

5.4 Simulations

In this context, the main role played by simulations is to facilitate the dialogue among analysts and stakeholders when checking the correctness of the formal specification. The formal specification can be animated with the simulation engine presented in Section 4.2. We customised the traces generated by the execution engine by defining functions that automatically translate the system state into a string that can be easily interpreted by humans. As an example of such a function, let us consider the theory for incidents' locations. Assume that the theory encodes the incident location with a natural number. In order to present the street name in a more human-readable format, a function (`street2string`) can be defined for converting numbers into actual street names. The function will be seamlessly used by the PVSio environment whenever printing the output —to this end, we exploit a PVS mechanism for defining automatic type conversions.

```
street_th: THEORY BEGIN %-- imports omitted
  street: TYPE = posnat
  street2string(s: street): string =
    COND s = 0  -> " Boulevard rd. "
         s = 1  -> " Terrace pl. "
          ... ENDCOND
 CONVERSION street2string
END street_th
```

The conversion can be defined for any PVS data type used in the system state, thus enabling a full customisation of the output. In the following, we show an example of simulation trace that can be obtained with the simulator. The simulator executes four simulation steps. For simplicity, here we have redefined the print function of the simulator so that it shows only the initial and final system states, and the sequence of actions performed.

```
<PVSio> exec(4)(LAS_task(sys)(initial_task_status), sys);
== Initial state =============
caller_cad( N/A )
caller_phone( N/A )
caller_location( N/A )
```

```
-------------------
incident_location( N/A)
incident_details( N/A )
==================================

>> ProQA gathers number and location <<
>> Call-Taker takes a call <<
>> Call-Taker verifies number <<
>> Call-Taker enters incident location <<

== Final State  =========
caller_cad( 1 )
caller_phone( +23 322 3860 843 )
caller_location((# latitude = 0.4W, longitude = 51.30N #))
-------------------
incident_location( Terrace pl. )
incident_details( N/A )
===================
```

6 Related Work and Conclusion

Tasks and work-flow analysis has been explored in several studies with different techniques and different aims. For instance, in [FUMK10], work-flows are initially modelled with a Web Service Business Process Language (WS-BPEL), and then such semi-formal models are translated into a Finite State Process (FSP) model suitable for verifying properties with model checking approaches; in [ZAW+10] and [DXW08], Petri nets based formalisms are used for modelling and analysing industrial and business processes. Doherty et al [DCH00] used PVS for defining an approach suitable for studying representational issues in distributed cognition systems.

In our work, on the other hand, the main goal is to build a formal specification that can be naturally mapped to the informal description of DiCoT information flow models. A DiCoT analysis of a system can have several ultimate aims, but whatever the overall aim, the immediate purpose is to draw out understanding and so create an accurate description of the distributed system. Our hope in providing a formalisation of DiCoT is that it would facilitate the creation of such a description with a non-ambiguous language that could be directly used by human-computer interaction experts for modelling the system.

Formalising the informal DiCoT information flow model for the London Ambulance Control room did prove insightful in several ways. Even the exercise of building a formal specification of the system and simulating it highlighted several issues, mainly due to under-specified procedures and hidden hypotheses. The original DiCoT analysts of the London Ambulance System agreed this was potentially useful in improving the analysis. Hence, even before using automated reasoning tools and techniques, the formal specification can be a suitable means for triggering questions to feed discussions on system design and ensure that the ultimate description obtained covers the important aspects of the system.

Acknowledgements: Funded as part of the CHI+MED: Multidisciplinary Computer-Human Interaction research for the design and safe use of interactive medical devices project, EPSRC Grant Number EP/G059063/1, and Extreme Reasoning, Grant Number EP/F02309X/1.

Bibliography

[BF06] A. Blandford, D. Furniss. DiCoT: A Methodology for Applying Distributed Cognition to the Design of Teamworking Systems. *Interactive Systems*, pp. 26–38, 2006.

[BMP08] C. Bernardeschi, P. Masci, H. Pfeifer. Early Prototyping of Wireless Sensor Network Algorithms in PVS. In Harrison and Sujan (eds.), *Proc. of SAFECOMP08*. Lecture Notes in Computer Science 5219, pp. 346–359. Springer, 2008.

[BMP09] C. Bernardeschi, P. Masci, H. Pfeifer. Analysis of Wireless Sensor Network Protocols in Dynamic Scenarios. In *Proc. of SSS09*. Lecture Notes in Computer Science 5873, pp. 105–119. Springer, 2009.

[BS98] R. Butler, J. Sjogren. A PVS Graph Theory Library. NASA Technical Memorandum 1998-206923, NASA Langley Research Center, Hampton, Virginia, 1998.

[CH97] J. C. Campos, M. D. Harrison. Formally verifying interactive systems: A review. In *Proc. of the 4th Intel. Eurographics Workshop on Design, Specification, and Verification of Interactive Systems*. Pp. 109–124. Springer, Berlin, 1997.

[COR+01] J. Crow, S. Owre, J. Rushby, N. Shankar, D. Stringer-Calvert. Evaluating, Testing, and Animating PVS Specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2001.

[DCH00] G. J. Doherty, J. C. Campos, M. D. Harrison. Representational Reasoning and Verification. *Formal Aspects of Computing* 12(4):260–277, 2000. ISSN: 0934-5043.

[DXW08] H. Dun, H. Xu, L. Wang. Transformation of BPEL Processes to Petri Nets. In *Theoretical Aspects of Software Engineering, 2008. TASE '08. 2nd IFIP/IEEE International Symposium on*. Pp. 166–173. June 2008.

[FB06] D. Furniss, A. Blandford. Understanding Emergency Medical Dispatch in terms of Distributed Cognition: a case study. *Ergonomics Journal* 49:1174–1203, 2006.

[FH91] N. V. Flor, E. L. Hutchins. Analyzing distributed cognition in software teams: a case study of team programming during perfective software maintenance. In *Empirical Studies of Programmers: Fourth Workshop*. Pp. 36–64. 1991.

[FUMK10] H. Foster, S. Uchitel, J. Magee, J. Kramer. An Integrated Workbench for Model-Based Engineering of Service Compositions. *Services Computing, IEEE Transactions on* 3(2):131–144, April/June 2010.

[Fur04] D. Furniss. Codifying Distributed Cognition: A Case Study of Emergency Medical Dispatch. 2004. MSc Thesis, UCLIC, UCL Interaction Centre.

[Hut95] E. Hutchins. How a Cockpit Remembers Its Speed. *Cognitive Science* 19:265–288, 1995.

[KM94] D. Kirsh, P. Maglio. On Distinguishing Epistemic from Pragmatic Action. *Cognitive Science* 18:513–549, 1994.

[MD08] J. McKnight, G. Doherty. Distributed cognition and mobile healthcare work. In *Proc. of BCS-HCI '08*. Pp. 35–38. British Computer Society, Swinton, UK, 2008.

[MM84] A. Movaghar, J. Meyer. Performability modelling with stochastic activity networks. In *Proc. of the 1984 Real-Time Systems Symposium*. Pp. 215–224. 1984.

[Muñ03] C. Muñoz. Rapid prototyping in PVS. Technical report NIA Report No. 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA, 2003.

[Nor02] D. A. Norman. *The Design of Everyday Things*. Basic Books, New York, reprint paperback edition, 2002.

[ORR+96] S. Owre, S. Rajan, J. Rushby, N. Shankar, M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In Alur and Henzinger (eds.), *Computer-Aided Verification, CAV '96*. Lecture Notes in Computer Science 1102, pp. 411–414. Springer-Verlag, New Brunswick, NJ, July/August 1996.

[Pri05] Priority Dispatch Corp. Inc. ProQA 3.4, Emergency Dispatch Software. 2005. www.prioritydispatch.net/support/pdf/ProQA_User_Guide.pdf.

[RBCB09] R. Rukšėnas, J. Back, P. Curzon, A. Blandford. Verification-guided modelling of salience and cognitive load. *Formal Aspects of Computing* 21:541–569, 2009.

[RE94] Y. Rogers, J. Ellis. Distributed cognition: an alternative framework for analysing and explaining collaborative working. *Journal of Information Technology* 9:119–128, 1994.

[Rus02] J. Rushby. Using Model Checking to Help Discover Mode Confusions and Other Automation Surprises. *Reliability Engineering and System Safety* 75(2):167–177, February 2002. Available at http://www.csl.sri.com/users/rushby/abstracts/ress02.

[SO99] N. Shankar, S. Owre. Principles and Pragmatics of Subtyping in PVS. In Bert et al. (eds.), *Proc. of WADT '99*. Lecture Notes in Computer Science 1827, pp. 37–52. Springer-Verlag, Toulouse, France, September 1999.

[SRSF06] H. Sharp, H. Robinson, J. Segal, D. Furniss. The Role of Story Cards and the Wall in XP teams: A Distributed Cognition Perspective. In *Proceedings of the conference on AGILE 2006*. Pp. 65–75. IEEE Computer Society, Washington, DC, USA, 2006.

[Thi08] H. Thimbleby. Ignorance of interaction programming is killing people. *ACM Interactions*, pp. 52–57, September/October 2008.

[ZAW+10] H. Zha, W. van der Aalst, J. Wang, L. Wen, J. Sun. Verifying workflow processes: a transformation-based approach. *Software and Systems Modeling*, pp. 1–12, 2010. 10.1007/s10270-010-0149-9.