Public Paper no. 136

Using PVS to Support the Analysis of Distributed Cognition Systems

Paolo Masci, Paul Curzon, Dominic Furniss & Ann Blandford

Masci, P., Curzon, P., Furniss, D., & Blandford, A. (2015). Using PVS to support the analysis of distributed cognition systems. *Innovations in Systems and Software Engineering, 11,* 113–130.

PP release date: 10 June 2013

Noname manuscript No.
(will be inserted by the editor)

Using PVS to support the analysis of distributed cognition systems

Paolo Masci · Paul Curzon · Dominic Furniss · Ann Blandford

Received: XXX / Accepted: YYY

Abstract The rigorous analysis of socio-technical systems is challenging, because people are inherent parts of the system, together with devices and artefacts. In this paper, we report on the use of PVS as a way of analysing such systems in terms of Distributed Cognition. Distributed Cognition is a conceptual framework that allows us to derive insights about plausible user trajectories in socio-technical systems by exploring what information in the environment provides resources for user action, but its application has traditionally required substantial craft skill. DiCoT adds structure and method to the analysis of socio-technical systems from a Distributed Cognition perspective. In this work, we demonstrate how PVS can be used with DiCoT to conduct a systematic analysis. We illustrate how a relatively simple use of PVS can help a field researcher (i) externalise assumptions and facts, (ii) verify the consistency of the logical argument framed in the descriptions, (iii) help uncover latent situations that may warrant further investigation, and (iv) verify conjectures about potential hazards linked to the observed use of information resources. Evidence is also provided that formal methods and empirical studies are not alternative approaches for studying a socio-technical system, but that they can complement and refine each other. The combined use of PVS and DiCoT is illustrated through a case study concerning a real-world emergency medical dispatch system.

Paolo Masci, Paul Curzon
Queen Mary University of London
School of Electronic Engineering and Computer Science
E-mail: {paolo.masci, paul.curzon}@eecs.qmul.ac.uk

Dominic Furniss, Ann Blandford
UCLIC, UCL Interaction Centre
University College, London
E-mail: {d.furniss,a.blandford}@ucl.ac.uk

Keywords Formal analysis, Higher Order Logic, PVS, Distributed Cognition, DiCoT, Socio-technical systems.

1 Introduction and background

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 error. 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 computer system failures and systematic human errors. The development process of a system of whatever level, therefore, must adopt appropriate means to eliminate design errors. This is especially relevant in safety-critical domains, such as healthcare.

In complex socio-technical systems, like hospitals or power plants, operators are required to follow written procedures that specify what actions must be taken to achieve intended goals. The best way of achieving a goal is usually context-dependent, but written procedures cannot cover all possible situations and contexts [38]. As a consequence, actual practice may deviate from written procedures (e.g., see [33]). Studying actual practice in a systematic and rigorous way is key to identifying potential hazards in such socio-technical systems, given trial-and-error approaches should be avoided in safety-critical contexts, and given operators should get work done correctly on the first attempt. Also, better understanding of current practice can help redesign a system in response to performance deviations and potential hazards.

Contextual studies such as ethnographic studies provide a way to understand how a system works in practice. They involve collecting information from people actually working in the system, including observing activities as they are carried out in the real workplace. This contrasts with studies based on experimental or simulated settings. A common difficulty in ethnographic studies is that field researchers have limited time and resources to perform the study. They therefore need to make decisions about what to look for and when. Distributed cognition [19] is a conceptual framework that can help field researchers make those decisions.

Distributed cognition is a conceptual framework proposed by Hutchins in [19]. It is based on the idea that cognition is not confined to the mind of humans, but spans across humans and artefacts. As such, cognition is distributed across the system, and can be described in terms of transformations of the representational state of information resources. Based on this view, Hutchins argues that it is possible to deduce important information about the cognitive activities of the users of a system by reasoning about the observable representational states of information resources. Furthermore, he observes that even if the cognitive activities of users remain hidden, in many cases the representational state of information and the kind of errors made by users impose constraints that are tight enough to enable an accurate identification of plausible internal representations and processes that the users must be adopting. An important implication of this is that the design of artifacts and technologies can be used not only for understanding plausible internal representations, but also for *shaping* them to ones that are "syntactically correct". This is meant in the sense that they can provide guidance to the people who have to perform tasks, thus making the path to achieving that task apparent.

Examples of activities that involve distributed cognition processes range from familiar everyday interactions to complex interactive systems. For example, in 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 — we change the environment to support the task we need to perform [21]. Similar considerations also apply in complex interactive systems, such as cockpits. For example, Hutchins, in [20], 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 [6] is a semi-structured methodology that has been proposed by the human-computer interaction community for applying distributed cognition to the analysis of teamwork systems. The approach has been successfully applied to analyse various real-world socio-technical systems (see for instance [37,27,39,33]). DiCoT proposes three core interdependent models to analyse socio-technical systems: the *physical model*, which studies the physical layout of the system; the *information flow model*, which studies how information is transformed and propagated in the system; the *artefact model*, which studies how artefacts are designed and used in the system. Associated with each of these models is a representation or diagram and distributed cognition principles. The DiCoT models will be described in detail in Section 4.

2 Contributions

In summary, this paper makes several novel contributions. We develop a constructive approach for using the specification and verification system PVS [31] in combination with DiCoT, an semi-structured analysis method for studying socio-technical systems from a distributed cognition perspective.

We present a set of generic PVS theories suitable for guiding the translation of DiCoT models into PVS higher-order logic specifications, and show how to use such theories to support rigorous reasoning about socio-technical systems design.

We demonstrate that a relatively simple use of PVS can help field researchers (i) automate various consistency checks for the DiCoT models, (ii) systematically challenge the logical argument they describe in the models, (iii) help uncover latent situations that may warrant further investigation, and (iv) verify conjectures about potential hazards linked to the observed use of information resources. Note that the proposed approach is not intended to replace the existing DiCoT analysis, but to complement it by using automated reasoning tools. In fact, we use PVS in such a way that it systematically challenges (through proof obligations) the reconstruction of facts and events observed by the field researcher.

We show that properties of interest can be automatically generated out of the specification, and that proof attempts can be automatically performed by the PVS system, reducing the perceived cost of using the tool. Whenever a proof obligation cannot be discharged (either automatically, or through an interactive proof) then a situation is found where the logical argument framed in the models may contain gaps that need to be filled (e.g., the field researcher is using hypotheses not explicitly stated in the model), or inconsistency that may warrant further investigation. A set of generic PVS theories is also presented to guide modelling and analysis. The theories can be animated with the PVSio extension of PVS, thus enabling the analyst to perform preliminary analyses through simulations.

The pragmatic use of PVS described in this paper explores new ways of using software tools during field studies. To date, software tools have been developed and used by field researchers to store and encode information efficiently — see for instance [40]. We provide evidence of the potential benefits of an integrated approach using both PVS and DiCoT by re-analysing field study data from a previously published case study with the original investigator. The considered case study focuses on an emergency dispatch system [17,18]. In section 6, we will provide a step-by-step analysis of the socio-technical system, and show that additional insights can be gained from the same data collected in the field study, providing therefore some evidence that the use of PVS allows a finer-grained analysis.

A preliminary version of this work was presented in [24], where we gave only an outline of the developed PVS theories to specify DiCoT information flow models, along with an overview of the insights that could be gained when using PVS in conjunction with DiCoT. An informal description of the constructive procedure has been presented in [23], and its application to support incident investigations has been explored in [26]. In [25], the approach has been successfully used within the context of a live field investigation.

2.1 Organisation of the paper

In Section 3, we provide an overview of PVS, focussing on the features of the tool relevant to this work. In Section 4, we present the DiCoT approach in detail. In Section 5, we present a method for supporting a DiCoT analysis (how to translate existing DiCoT models, how to analyse them in a such a way that insights can be gained about how to refine the analysis) and a set of generic PVS theories for guiding the specification and analysis. In Section 6, we apply the proposed method and developed PVS theories to analyse DiCoT models developed in an already performed and complete study. The considered example is based on the London Ambulance Service. In Section 7, we discuss related work. In Section 8, we draw conclusions.

3 Prototype Verification System (PVS)

PVS [31] is a specification and verification system that combines an expressive specification language with an interactive proof checker. The PVS specification language is based on strongly typed higher-order logic, which allows quantification over propositional functions to be formulated. The language includes 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* [36], which is a powerful mechanism for expressing complex consistency requirements. An example of a subtype is {`x: real | x` $\neq$ `0`}, which is derived from real numbers by using the predicate `x` $\neq$ `0`. When using expressions with subtypes, PVS automatically generates proof obligations, denominated type correctness conditions (TTCs), for ensuring the valid use of the type. We will rely on this automatic generation of proof obligations to check the consistency of the logical argument framed in the DiCoT models.

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 has a pre-defined built-in prelude, and a variety of loadable libraries, such as the NASA library [11], which provides several standard definitions and proved facts that can be used when developing new theories.

PVS includes an automated theorem prover that can be used to interactively apply powerful inference procedures within a sequent calculus framework. The primitive inference procedures include, among others, propositional and quantifier rules, induction, simplification using decision procedures for equality and linear arithmetic, data and predicate abstraction [31].

PVS has a ground evaluator [12] 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 [30], 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 PVS theorem prover.

4 Distributed Cognition for Teamwork (DiCoT)

In this section, we report a detailed description of the DiCoT models we will consider: physical, information flow, and artefact. For each model, we describe (i) what elements should be represented in the model, (ii) what analysis should be performed on the model, and (iii) the characteristics of a suitable specification language for the models.

Physical model. A DiCoT physical model describes the structure of the physical layout of the environment where actions are performed by individuals.

The model highlights what media can be used by individuals to extend their cognitive space (e.g., artefacts, information technologies, and other individuals) and what information resources can be held by such media (e.g., written notes, displayed data, utterances).

The analysis of the physical models aims to identify constraints and affordances provided by the physical environment. In [17], two strategies are suggested for choosing the bounds of the environment to be modelled: individual- and location-based. The individual-based strategy identifies the bounds of the environment with what individuals can physically hear, see, or have access to while actions are performed. Depending on the aim of the analysis, this perspective can be extended from an individual perspective to a team, a working unit, or an organisation perspective. The location-based strategy identifies the bounds of the environment on the basis of physical locations, e.g., a desk, a room, a floor or a building. A suitable modelling language for these models should allow the layout structure and the physical location of actors and artefacts to be specified.

Information flow model. A DiCoT information flow model describes how information resources are transformed and propagated in the system.

The model highlights the role of users, technologies and artefacts during communication, and describes the sequence of events observed in the system.

The analysis of the information flow models aims to (i) identify relevant aspects and potential issues related to single tasks and communications, and (ii) assess system-wide properties emerging from the way information resources are processed, e.g., identification of information buffers, where a person withholds information temporarily from someone else until they naturally pause from concentrating on a different task. In [17], a three-level hierarchical modelling approach is presented to aid investigators build the model. The first level aims to model a high-level summary of the function of the system; as such, the developed model describes the main purpose of the system, the information going into the system, and the system output. The second level aims to model in detail the routine activities carried out in the system by highlighting the task and the communications between team members. The third level aims to model events and cases that break the routine activities. Each developed model includes a narrative description and a number of ad hoc semi-formal diagrams for understanding how the system works and for reasoning about design issues. A suitable modelling language for these models should allow us to identify activities and dependency relations (such as temporal ordering) among activities.

Artefact model. A DiCoT artefact model describes in detail how specific media and information resources are used within the system.

This model can be seen as a refinement of the DiCoT physical and information flow models, as it provides additional details about the structure of selected artefacts and about their use in the system. In particular, the artefact model aims to highlight the role of artefacts in tasks and how their interface facilitates or hinders work. Artefacts are selected on the basis of those aspects of the system that are deemed central for its function, e.g., what a field study researcher believes important for understanding how actual practice is carried out, or what a designer believes important for studying how a design change may affect the socio-technical system.

In [17], the Resource Model [41] is suggested as a starting point to study how information resources are distributed in the system, either internalised in the human mind or externalised in the environment. In the Resource Model, six abstract information structures are used to classify information resources: plans (sequences of actions that could be carried out), goals (system states that one would like to achieve), affordances (sets of possible next actions that can be taken from the current system state), history (lists of actions already taken to achieve the current system state), action-effect (a transition relation that defines how the current state is transformed when an action is taken), and current state (a collection of relevant values of objects in the environment). A suitable modelling language for the artefact model should enable a detailed specification of the structure and transformations of information resources at different levels of detail.

5 Using PVS to support a DiCoT analysis

In this section, we illustrate a constructive procedure for building PVS theories suitable for supporting a DiCoT analysis. The proposed procedure is general and represents a guideline for specifying and analysing in PVS: field study data[1], existing DiCoT models, and

[1] Field study data considered here consist of written notes produced by the field investigator during the observations.

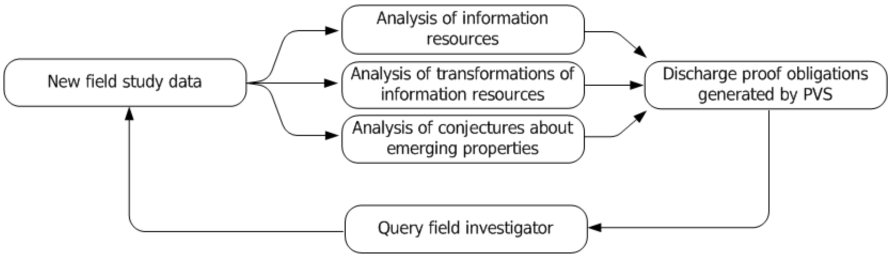
New field study data
Analysis of information resources
Analysis of transformations of information resources
Analysis of conjectures about emerging properties
Discharge proof obligations generated by PVS
Query field investigator

Fig. 1 Overview of the approach.

user manuals. A set of generic PVS theories to support the constructive approach are presented in sections 5.1 5.2 5.3, and their application is shown in section 6. An overview of the approach is depicted in Figure 1. In the following, we illustrate the approach.

Analysis of information resources. *The first step* is to develop PVS theories that allow field researchers to externalise facts about information resources available from artefacts, technologies and individuals (e.g., information printed on labels or displayed by devices). Each information resource is modelled using a different PVS data-type. The type definitions can be given at different level of details, according to what the field researcher deems relevant. Uninterpreted types are used to model information resources at the highest level of abstraction — uninterpreted types just define the name of information resources. Known constraints about information resources are embedded in the type definitions through *predicate subtyping*. The utility of using predicate subtypes is that PVS automatically generates proof obligations and proof attempts to demonstrate the consistency (or otherwise) of the specification, discharging therefore field researchers from the burden of specifying (and manually checking) consistency constraints. When a proof obligation cannot be discharged, either automatically or by guiding the theorem prover, then a situation is found that may warrant further investigation — e.g., the field researcher is using assumptions not explicitly stated in the model; the model specification contains inconsistencies/gaps; or there is an actual issue related to the information resources provided by the real-world system. Examples of constraints that can be expressed by subtyping are the physical location of individuals in indoor settings (e.g., how desks and walls limit movement and communication), physical characteristics of artefacts (e.g., the number of post-its on a board), limits of observable information externalised by devices (e.g., the maximum numbers of calls that can be displayed on a digital display board).

Analysis of transformations of information resources. *The second step* is to develop PVS theories that allow field researchers to specify how information resources are generated, transformed and propagated in the system. In PVS, we specify such transformations as transition functions over system states. As for information resources, transformations can be provided at different levels of detail, according to what the field researcher deems relevant to the analysis. For instance, the field researcher can be very detailed about the use of some artefacts and information technologies (e.g., describing how operators organise paper reports on a desk, or how a specific report's field is entered in an electronic record) or simply describe abstract relations between individuals (e.g., which communication channels exist between operators). The PVS specification language allows one to use different levels of details — as for information resources, uninterpreted types can be used to specify abstract relations, and subtypes and actual definitions can be used to gradually add detail. The system state is given by the observable state of information resources. Predicate subtyping can be used to express constraints about the possible transformations allowed by the system — PVS will automatically generate proof obligations to ensure that the transformations are used consistently within the specification.

Analysis of conjectures about emerging properties. *The third step* is to develop PVS theories for checking conjectures about why physical layouts and work-flows are the way they are, and for identifying potential hazards linked to the observed use of information resources. Conjectures can be specified as predicates over information resources and over transformations of information resources. The analysis of potential hazards can be supported by a systematic formalisation of user manuals and written procedures, as they provide insights about the designers' perspective on the system. This kind of analysis is useful to pull out potential hazards linked to latent situations. The approach has similarities with the analysis carried out by Rushby in [34, 2]. 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. He argues that a strong divergence between the mental model and the actual behaviour prescribed by the interactive system may lead to *automation surprise*, i.e., situations where the automated system behaves in a way that is different from that expected by the operator. In our case study, user manuals and written procedures are used as the basis for specifying the behaviour of the interactive system, and the DiCoT models are used to derive insights about plausible mental models developed by users of the system.

5.1 Generic PVS theories to support DiCoT modelling

In the following we present a set of generic PVS theories developed with the aim of guiding analysts during the specification process and help reduce the perceived cost of using PVS in a DiCoT analysis. The developed theories allow one to build a PVS specification which can be *naturally mapped* to the DiCoT models, i.e., the relationship between the DiCoT models and the PVS specification can be easily seen and justified. The specification style used in the generic PVS theories allows PVSio to be used to animate the specifications, thus can potentially facilitate interdisciplinary discussion about the situations described in the models. In section 5.3, we present a simulation engine developed on top of PVSio that automates the generation of simulation traces out of the PVS specification of the DiCoT models.

System State. Information resources are specified as fields of a record type, `system_state`. Each information resource is characterised by a unique resource identifier. Resource identifiers can be specified either explicitly (e.g., through natural numbers or enumerated types), or implicitly (through the type name).

The level of detail at which an information resource is modelled generally depends on the aim of the DiCoT analysis. At the highest level of abstraction, fields are specified as uninterpreted types. In order to enable a modular construction of the specification, we exploit *information hiding* when defining theories for the system state. Each PVS theory is assimilated to the class concept used in object-oriented programming languages: interface functions are used for accessing and modifying data types in a consistent way, and the actual specification of the data-types is hidden.

Activities. Activities are actions carried out within the system by individuals or information technologies. They are specified as transition functions over system states. Each activity has a unique identifier. We developed a PVS theory, `activity_th`, to provide some standard type 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; and 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 the system state and the number of activities are theory parameters, and must be instantiated when importing the theory.

```
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. Tasks define dependency relations among activities, which can be performed either in sequence or concurrently. We specify tasks with a graph-based notation suitable for formalising a graphical notation used in DiCoT: nodes in the graph represent activities, and edges between nodes represent dependency relations between activities. Drawing concepts from Activity Networks [29], a widely used formalism for modelling complex concurrent systems, we associate an *enabling predicate* to each activity, which can be specified as predicates over system states. An enabling predicate defines the necessary pre-condition for performing the associated activity. When an enabling predicate is *true*, we say that the associated activity is *enabled*. By default, an activity is enabled when all directly connected activities have already been performed. Whenever an activity becomes enabled, the activity can be performed. When several activities are enabled at the same time, then such activities can be performed concurrently. Activities are atomic: the concurrent execution of activities is therefore a sequential execution where any ordering is possible. When an activity is performed, the system state is updated according to the function defined by the activity, and we say that the activity *completes*. Dependency relations among activities can be parametric with respect to *control flow conditions*, which define different ways of continuing the task. If a control flow condition is *true*, then the associated edge is included in the graph describing the task; otherwise, the edge is removed. Control flow conditions can be specified as predicates over system state associated to edges (hereafter, *dependency predicates*).

In PVS, we therefore specify tasks as structured data types consisting of five fields: `F`, a function that associates a unique identifier to each activity that can be performed in the task; `G`, a directed graph that defines dependency relations among activities (the identifier of each node in the graph is given by the identifier of the associated activity); `P`, a function that associates dependency relations to dependency predicates; `S`, a status vector that defines the progress status of each activity in the task; `E`, a function that associates enabling predicated to activities (the default predicate checks that all activities connected with incoming edges are completed). The type definition of `G` uses the NASA library on directed graphs [11] 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, activity progress status, and enabling predicate are theory parameters, i.e., they are left unspecified, and must be instantiated by the theories importing `task_th`.

```
task_th[system_state, activity, activity_id,
       progress_status, enabling_predicate: 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],
       E: [activity_id -> enabling_predicate] #]
END task_th
```

5.2 Generic PVS theories to support DiCoT analysis

We have developed a generic PVS theory to provide a template for comparing specifications of activities derived from different sources. The aim is to identify unsafe divergence between actual practice (as described in a DiCoT model) and prescribed practice (as described, for instance, in user manuals). Informally stated, unsafe divergences are situations where critical information resources have strongly different values. An example of unsafe divergence in an emergency dispatch system is that an ambulance is supposed to receive the emergency address when an operator enters the emergency address in the information system (e.g., according to the specification derived from field study data), while in practice he/she doesn't (e.g., according to the user manual). The precise definition of "unsafe divergence" depends on the considered situation — it may change from system to system, and over time.

A generic approach suitable for developing a tailored specification of unsafe divergence can be obtained through the use of an abstract domain. In the abstract domain, we can define equivalence classes reflecting the concerns that are deemed safety critical for the system, and thus map actual values of information resources to those abstract classes. By using the abstract domain, we can say that a system is not in a situation of unsafe divergence (or, alternatively, say that the system is operating within its safety limits) as long as information flows described in the specifications we are comparing generate equivalent abstract states.

More precisely, given any pair of corresponding activities from two specifications, if we start from two concrete states that are equivalent in the abstract domain, then the system is within safety limits if the new states obtained by executing the activities are still equivalent in the abstract domain.

In PVS, we can support the analysis described above by defining a data type for the abstract domain, and an abstraction function for mapping system states from the concrete specification into the abstract domain. To this end, we defined a parametric PVS theory, `safe_divergence_th`, which defines a predicate, `safe_divergence`, for specifying claims about divergences in terms of five parameters: the state of the abstract domain (`abstract_state`), which specifies what information is deemed safety-critical for the system; two function types (`actual_state`, `pre scribed_practice`), which specify a pair of concrete transitions that we want to compare; two function types (`alpha1`, `alpha2`) defining the relations to abstract concrete system states into abstract states.

```
safe_divergence_th[
   safety_state, actual_state,
   prescribed_state: TYPE,
   alpha1 : [actual_state -> safety_state],
   alpha2 : [prescribed_state -> safety_state] ]:
THEORY BEGIN
 safe_divergence?
  (st1: actual_state, st2: prescribed_state)
  (f1: [actual_state -> actual_state],
   f2: [prescribed_state -> prescribed_state]):
     bool
  = (alpha1(st1) = alpha2(st2))
         => (alpha1(f1(st1)) = alpha2(f2(st2)))
END safe_divergence_th
```

5.3 Generic PVS theories to support model animation

We have developed a simulation engine for animating the formal specification of DiCoT information flow and artefact models. In this work, the main utility of the simulation engine is to facilitate the dialogue between field researchers, practitioners, and analysts when checking the correctness of the formal specification.

The developed simulation engine schedules activities according to the dependency relations specified in a task, and uses PVSio for generating visual feedback of the execution. Drawing ideas from approaches for the analysis of protocols for distributed systems of autonomous and cooperating nodes [4,5], we define the simulation engine as a higher-order function, `simulate`, that iteratively selects, through a *scheduler*, the activity to be performed. The iteration is performed at most `N` times. At each iteration a new system state is generated by applying the transition function of the selected activity to the current system state.

The scheduler identifies the activity to be executed on the basis of its progress status, 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 is enabled but it cannot be executed (this may happen because of control flows); and `deleted`, i.e., the activity has not been performed. Activities are chosen non-deterministically from two worklists, `C` and `R`. Worklist `C` contains cancelled activities. Worklist `R` contains activities ready for execution. Activities in worklist `C` have priority over those in worklist `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 worklist, the activity is also removed from the worklist. Two auxiliary functions, `update_completed` and `update_deleted`, are used in the engine for identifying the set of activities that become enabled after an activity completes. In particular, when activity i completes, function `update_completed` changes to `ready` the status of all activities j connected to i if the enabling predicate if j becomes *true* and the dependency predicate on edge (i, j) is *true*. Otherwise, if the enabling predicate of j is *true* and the dependency predicate on (i, j) is *false*, the function changes the status of j to `cancelled`. Function `update_deleted`, on the other hand, changes the status of cancelled activities to `deleted`. The system state is left unchanged when an activity is deleted.

Function `simulate` uses function `execute` defined in `activity_th` for generating the new system state when an activity completes. The execution of the task terminates either when function `simulate` has been executed `N` times, or when both worklists `R` and `C` are empty. The PVS specification of function `simulate` follows.

```
simulate(N: nat): RECURSIVE
[task, system_state -> system_state] =
  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
```

6 PVS-aided analysis of the London Ambulance Service

In this section, we apply PVS to the analysis of representative DiCoT models developed in the context of an already performed and completed field study [17, 18]. The considered field study investigated the activities carried out in the Central Control Room of the London Ambulance Service (LAS). The LAS Central Ambulance Control room 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 (and how many ambulances) should be allocated to which incident. In the dispatching area there is a sector desk for each zone of the city (London has seven zones), plus a fast response unit (FRU), which responds to urgent emergencies (red calls), a helicopter emergency medical service desk (HEMS), an administration desk, which provides support services (e.g., vehicle maintenance), and a control desk, which supervises the operation of the whole room.

In the following, we describe step-by-step how PVS can be used to support the DiCoT analysis. Following the approach described in section 5, we formalise the considered DiCoT models in PVS, and use PVS to automate the analysis. We will show that additional insights can be derived from the same models analysed in [17], providing evidence of the benefits gained when using PVS to systematically check details at a finer-grain. In the considered examples, we will use a level of detail appropriate for the purposes of this article. Readers interested in a comprehensive description of the Central Control room should refer to [17].

6.1 Supporting DiCoT physical layout analysis

We consider here a DiCoT physical model developed for studying how the physical layout affects the activities carried out in the dispatching area of the LAS Central Ambulance Control room.

Physical layout of the dispatch room. The dispatch room has seven sector desks, each of which is responsible for allocating ambulances in an area of London. All sectors cooperate for ensuring an overall efficient service. In order to support co-operation, sector desks are organised such that they reflect the geographical location of sectors (see Figure 2). This organisation of sector desks aims to ease direct face-to-face communication between allocators — some incidents may require cross-sector coordination. The fast response unit has a central position in the room layout because the unit aims to support all other desks for urgent emergencies (red calls). The room contains two display boards, which report the amount of incoming calls waiting to be answered, and the percentage of calls that has been answered within given pre-defined time-frames. The room also contains a city map.

PVS-aided analysis. We aim to specify in PVS the concepts framed in the DiCoT physical model. To this end, we define three different data-types for specifying the characteristics of displays, sector desks and maps described in the DiCoT model. Displays are specified as a record type, `display_board`, which contains two fields (one for each information resource provided by the display): `incoming_calls`, a bounded natural number, and `answered_calls`, a percentage. Desks in the control room are specified as an enumerated type, `CAC_desk`, which defines unique names for each desk. The map of the city is specified as an uninterpreted type, `map`, because additional details about it are not available from the field study data — the field researcher deemed the map not relevant for the specific analysis. The system state is thus specified as a record type, `system_state`, which contains a finite set of desks (`desks`), two display boards (`display_a` and `display_b`), and a city map (`city_map`).

```
CAC_resources_th: THEORY
BEGIN
 MAX_CALLS    : posnat
 percentage   : TYPE
   = {x: real | x >= 0 AND x <= 100}
 display_board: TYPE
   = [# incoming_calls: upto(MAX_CALLS),
        answered_calls: Percentage #]
 CAC_desk     : TYPE
   = { NW_desk,  W_desk, SE_desk, SW_desk,
       EC_desk,  C_desk, NE_desk, HEMS_desk,
       FRU_desk, Control_desk,    Admin_desk }
 map          : TYPE
 system_state: TYPE
   = [# desks     : finite_set[CAC_desk],
      display_a : display_board,
      display_b : display_board,
      city_map  : map #]
END CAC_resources_th
```

The specification above can be refined by including additional facts about information resources. From the observations carried out in the field study, for instance, we know that the display boards are updated when new calls are received. This fact can be specified with a function, `new_incoming_call`, which increments the number of incoming calls on the displays. Initially, we can use an uninterpreted function, and use predicate subtyping to assert that the function returns a new state with a larger number of incoming calls.

```
new_incoming_calls(sys: system_state):
   {s: system_state |
        incoming_calls(display_a(s))
          > incoming_calls(display_a(sys))
       AND incoming_calls(display_b(s))
               > incoming_calls(display_b(sys))}
```

With the above specification, PVS automatically generates an *existence TCC*, i.e., PVS requires evidence that a function can be implemented that is compliant with the uninterpreted function type specification. The simplest implementation we could use for discharging the proof obligation is a function that increments the number of incoming calls by 1:

```
LAMBDA(sys: system_state): sys WITH
 [ display_a :=
    (# incoming_calls
         := incoming_calls(display_a(sys)) + 1,
       answered_calls
         := answered_calls(display_a(sys)) #),
   display_b :=
    (# incoming_calls
         := incoming_calls(display_b(sys)) + 1,
       answered_calls
         := answered_calls(display_b(sys)) #) ]
```

When using the function above in the proof attempt, the interactive theorem prover automatically identifies a situation that violates another type constraints:

```
 |-------
{1} FORALL (sys: system_state):
     1 + incoming_calls(display_b(sys))
            <= MAX_CALLS
```

The above situations points out that the proposed function is not guaranteeing a constraint for the maximum value that can be shown on the display. Although mathematically trivial, this violation highlights issues that may be glossed over in an informal description but may warrant further investigation: What is the maximum number that can be shown on the display boards? What happens if the maximum number is reached? The first

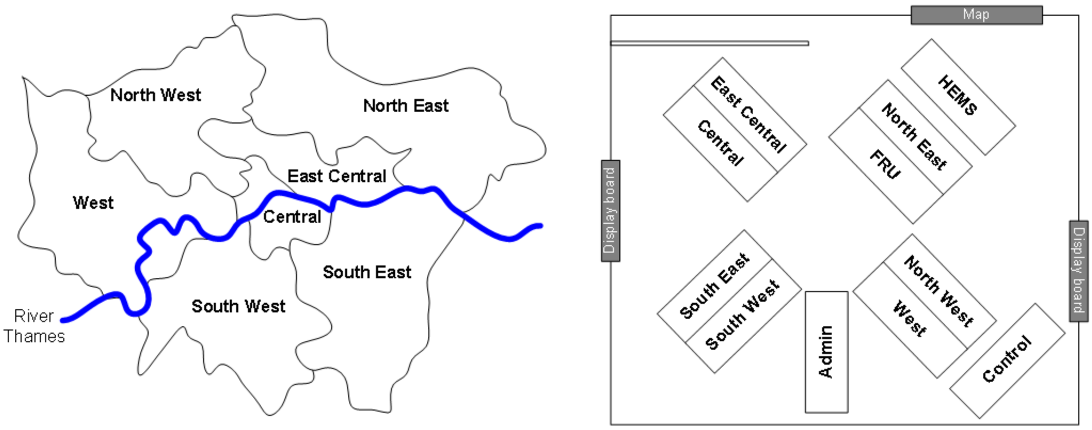

Fig. 2 Geographical sectors in London and schematic diagram of the LAS Central Ambulance Control room layout.

question reflects the concern that, in any situation, the maximum number should never be reached. The second question reflects concerns about critical situations when the system is pushed to its limits.

Further support from PVS can be obtained by formulating claims about emergent properties. In this case, the claim we consider is about the relation between the geographical position of sectors and the physical location of sector desks: physically close sector desks correspond to adjacent geographical sectors on the map. In order to formulate this claim, we define: an enumerated type, `sector_geo`, which specifies a set of unique identifiers for the sectors; type `sector_desk`, a subtype of `desk` that identifies sector desks in the dispatching area; a transformation function, `sector`, which specifies the relation between sector desks and geographical sectors; two predicates, `adjacent?` and `f2f_communication?`, which specify what sectors are geographically adjacent and what is to be considered a face to face communication.

```
sector_geo : TYPE = { NW, W, SE, SW, EC, C, NE }
sector_desk: TYPE = { d: CAC_desk |
                          d /= Control_desk
                           AND d /= Admin_desk }
sector(s: Sector_desk): finite_set[sector_geo] = %..
adjacent?(s1, s2: finite_set[sector_geo]): bool %..
f2f_communication?(s1, s2: sector_desk): bool %..
```

With the definitions above, we can then specify the claim in PVS as follows:

```
sector_desk_claim: CLAIM
 FORALL (d1, d2: Sector_desk):
  f2f_communication?(d1, d2)
    => adjacent?(sector(d1),sector(d2))
```

The claim can be automatically verified in seconds with PVS through one of its automated decision procedures, e.g., `grind`. We note here that the provided example aims to demonstrate how emergent properties can be specified and verified in PVS. Although the specific example is simple, its utility is to allow one (i) to externalise concepts that could otherwise lie hidden in the analyst's head, and potentially lead to different interpretations and understanding from others, and (ii) to systematically check basic claims in a completely automatic fashion — when these check fails, the theorem prover points out the specific situation that may require further investigation.

6.2 Supporting DiCoT information flow analysis

We consider here a DiCoT information flow model developed for studying in detail the activities carried out in the call taking area and the dispatching area of the Central Ambulance Control room. To support the analysis, we formalise some relevant paragraphs of the user manual of the computerised version of the emergency medical dispatch system used in the call taking and dispatching areas, and use PVS to compare the two specifications. The aim of the comparison is to systematically identify potential unsafe divergence between what has been observed (which reflects *actual practice*) and what is required by the information system according to the manual (which reflects *prescribed practice*).

Actual practice. 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 classify incidents in terms of their medical urgency. ProQA [32] is a computerised version of the system, and is currently used in the central ambulance control room. ProQA structures the dialogue between call-takers and external callers, and enables communication between call-takers and allocators. The protocol is the following. Initially, the call-taker greets the external caller and verifies the caller's location and telephone number both of which are automatically gathered by the ProQA system. Then, the call-taker starts a questioning procedure to distill information about the incident location and the complaint. As soon as the call-taker enters the incident 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. 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. Allocators can view the position of all ambulances. In the case of a new incident, the allocator mentally selects an ambulance on the basis of its location and availability status. As soon as the incident priority is known, the call-taker can provide support and advice to the caller, and the allocator can alert the ambulance crew and co-ordinate with it. If the ambulance crew accepts the mission, the allocator transfers the incident's details to the ambulance crew. Otherwise, the allocator transfers the incident's details to another allocator of a neighbouring sector.

User Manual. The ProQA application starts with a log-in screen — a log-in name and password should be assigned to each call-taker by the system administrator. After the log-in screen, the call-taker is presented with the *waiting for next incident* screen (page 2 of the ProQA user manual [32]). A computer aided dispatch (CAD) number is automatically assigned to the new incident. If needed, the call-taker can change the CAD number through the *change case number* function provided by ProQA. When starting a new case, the call-taker needs to enter the following information in a sequential order in the *case entry* screen (pages 71—81 of the ProQA user manual [32]): address of the emergency, which must be verified by having the caller repeat it; phone number, which must be verified by having the caller repeat it; the caller's name; a brief description of what happened (e.g., chest pain); whether the caller is with the patient (default: yes); the number of the injured person (default: 1); the age, either in months or years, of the injured person (default unit: years); the gender of the injured person (default: male); whether the injured person is conscious and breathing (default: yes). ProQA automatically identifies a chief complaint code reflecting the information entered by the call-taker. A *case timer* keeps track of the total elapsed time since when the call-taker started entering information about the incident. After the case entry screen, ProQA presents the *key question* screen to the call-taker (pages 71—81 of the ProQA user manual [32]), which specifies a set of additional questions pertinent to the incident that need to be asked of the caller (e.g., whether the caller is safe or in danger). ProQA automatically displays a *send dispatch* screen as soon as it has enough information to recommend a dispatch code (pages 88—90 of the ProQA user manual [32]). This may happen at any instant: either during the questioning process or after all questions have been answered. In order to deliver the dispatch code to allocators, call-takers must use the *send dispatch code* function provided by ProQA. The dispatch code automatically selected by the application can be changed by call-takers. Also, call-takers can delay sending the dispatch code when they believe it is appropriate to ask additional information of the caller.

PVS-aided analysis. We aim to specify in PVS the information flows described in the DiCoT model and in the user manual. To this end, we define two theories, `observed_state_th` and `user_manual_state_th`, for specifying the structure of the system state and the set of activities described in the DiCoT model and in the ProQA user manual.

According to the description provided by the DiCoT model, the system state can be specified as a record type containing two fields: `call_taker_info`, which defines the state of information resources handled by call-takers; `allocator_info`, which defines the state of information resources handled by allocators.

```
%-- system state
system_state: TYPE
 = [# call_taker_info: call_taker_state,
      allocator_info : allocator_state  #]
```

The call taker state is a record type containing three fields: `caller_phone`, an enumerated field type that specifies the phone number of the caller; `caller_location`, an enumerated field type that specifies the caller location; `incident_info`, of type `incident_state`.

```
%-- call taker state
call_taker_state: TYPE
  = [# caller_phone   : phone_number,
       caller_location: location_state,
       incident_info  : incident_state #]
```

The incident state is specified with three fields: `location`, an enumerated type field that specifies the location of the incident; `details`, a Boolean type field that specifies whether details about what happened are available; `priority`, an enumerated type field that specifies the priority of the incident.

```
%-- incident state
incident_state  : TYPE
  = [# location: location_state,
       details : boolean,
       priority: priority_code #]
```

Similarly, the allocator state is a record type with two fields: `incidents`, a vector of incidents; `ambulances`, a vector of elements of type `ambulance_info`, which specifies location and availability of each ambulance in the sector.

```
%-- allocator state
allocator_state: TYPE =
 [# incidents :
      [incident_id  -> incident_state],
    ambulances:
      [ambulance_id -> ambulance_state] #]
```

Transformations of information resources described in the DiCoT model are specified as transition functions over system states. In the following, we describe in detail the specification for the first two activities performed by call takers: the call taker takes a call (`ct_takes_call`), and the call taker verifies the call (`ct_verifies_call`). The specification of the other activities is only informally described here — the complete PVS specification will be made available at [1].

The first activity is `ct_takes_call`, which specifies the state transformation when the call taker takes a new call: location and telephone number fields become available. At the considered level of detail, the only information we have about the caller's phone and location is that the call taker will enter them. The uncertainty about whether the entered number has been properly validated can be specified through the `epsilon` function defined in the PVS prelude library, which nondeterministically chooses a value from the set provided as argument to the function. The specification follows.

```
%-- activity 1: call taker takes a call
ct_takes_call(st: system_state): system_state
= st WITH
   [ call_taker_info := call_taker_info(st)
     WITH
   [ caller_phone
      := epsilon({x: phone_number | x /= NA}),
          caller_location
      := epsilon({x: location_state | x /= NA})]]
```

The second activity is `ct_verifies_call`, which corresponds to the call-taker verifying the caller's location and telephone number. This transformation changes the call-taker state as follows: the caller's phone and location are verified, and the incident location is also verified. The function also modifies the allocator state — according to the description, ProQA automatically activates the relevant sector desk: field `allocator_info` of the allocator state will contain the incident location. The specification follows.

```
%-- activity 2: call taker verifies call
ct_verifies_call(incident: below(MAX_INCIDENTS))
               (st: system_state): system_state
 = LET new_incident: incident_state
        = call_taker_info(st)‘incident_info
           WITH [ location := verified_loc ]
    IN st WITH [
     call_taker_info := call_taker_info(st)
       WITH [ caller_phone    := verified_num,
              caller_location := verified_loc,
              incident_info   := new_incident ],
     allocator_info := allocator_info(st)
       WITH [ incidents := LAMBDA(x: incident):
                              IF x = incident_id
                               THEN new_incident
                              ELSE allocator_info(st)
                                    ‘incident(x)
                                        ENDIF ]]
```

Following the same approach, we formalised also the following activities: the call-takers enters the incident details (`ct_enters_details`); the call-taker provides support and advice to the caller (`ct_provides_support`); the allocator checks whether the incident is new or it has been already reported (`al_checks_incident`); the allocator mentally selects an ambulance (`al_selects_ambulance`); the allocator alerts the selected ambulance (`al_alerts_ambulance`); the allocator transfers the incident details either to the selected ambulance or to another allocator (`al_transfers_details`).

The concepts described in the ProQA user manual can be formalised in a similar way. The type definitions in the PVS specification will reflect the constraints imposed by ProQA. Namely, we define a new PVS datatype, `text_field`, for specifying the content of input boxes; the data-type has two constructors (`text` for completed input boxes, and `NA` for empty input boxes). We use enumerated types for selection boxes, and record types for information resources with fields. Information resources available to call takers are specified as a record type, `proQA_call_taker_state`, which, according to the user manual, has twelve fields:

`cad_number` a bounded natural number that defines the computer aided dispatch number automatically assigned by ProQA to identify incidents

`em_address` a text field that identifies the location of the emergency

`ph_number` a text field that identifies the caller's telephone number

`name` a text field that identifies the caller's name

`description` a text field that describes the kind of incident

`with_patient` an enumerated type that specifies if the caller is with the injured person — possible options are: yes, no, the patient himself/herself (first_party), a person that is directly involved with or in close proximity to the patient (second_party), a person that is not directly involved with or in close proximity to the incident (third _party), someone from a public service agency (fourth_party)

n_patients a natural number that specifies how many persons are injured

patient_age a composite field with a text box and a selection box for specifying the age of the injured person; we model this field be defining a new datatype, proQA_age

patient_gender an enumerated type for specifying the gender of the injured person (possible options are: male, female, unknown)

is_conscious an enumerated type for specifying if the injured person is conscious (possible options are: yes, no, unknown)

is_breathing an enumerated type for specifying if the injured person is breathing (possible options are: yes, no, unknown (third or fourth party caller who doesn't know if the patient is breathing), uncertain (second party caller who is uncertain if the patient is breathing), ineffective (the patient in unconscious and breathing is irregular or slow)

complaint_code a bounded natural number identifying the chief complaint

elapsed_time information about the time elapsed since when the new call has been taken; this is specified with a new data type, proQA_time.

The specification of the call taker state follows.

```
%-- call taker state
proQA_call_taker_state: TYPE =
    [#  cad_number     : below(MAX_CAD),
        em_address     : text_field,
        ph_number      : text_field,
        name           : text_field,
        description    : text_field,
        with_patient   : proQA_with_patient,
        n_patients     : nat,
        patient_age    : proQA_age,
        patient_gender: proQA_gender,
        is_conscious   : proQA_is_conscious,
        is_breathing   : proQA_is_breathing,
        complaint_code: below(MAX_COMPLAINT_CODE) #]
```

Information resources available to allocators are simplified by considering only three fields, as this is sufficient to show some additional insights further below. The considered fields are: the finite set of CAD numbers associated to incidents (cad_numbers), and two vectors of text fields reporting the emergency addresses (em_addresses) and the incident details (em_details).

```
%-- (simplified) allocator state
proQA_allocator_state: TYPE
 = [# cad_numbers : finite_set[below(MAX_CAD)],
      em_addresses: [below(MAX_CAD) -> text_field],
      em_details  : [below(MAX_CAD) -> text_field]#]
```

Given the specification above, we can then specify the activities described in the user manual as transition functions over states of type proQA_system_state.

```
%-- system state
proQA_system_state: TYPE
 = [# call_taker_info: proQA_call_taker_state,
      allocator_info : proQA_allocator_state #]
```

In the following, we describe in detail two activities which raise issues that may warrant further investigation. The specification of the other activities is available at [1]. The first activity we consider here is when the call-taker enters information about the gender of the patient. The activity is specified as a function proQA_patient_gender, which updates the call taker state by setting a gender (given as function parameter). The description provided by the user manual when entering the patient's gender states that *"If you enter Unknown as the answer to the age question and there is only one patient, ProQA allows you to continue with the questioning [by entering the patient's gender]."* (page 74 of the ProQA user manual [32]). We use a predicate subtype to model this constraint — we restrict the domain of the function to system states where either the age of the patient is known, or the number of injured persons is 1.

```
%-- activity 2: call taker enters patient age
proQA_patient_gender(gen: proQA_gender)
 (st: {st: proQA_system_state |
        call_taker_info(st)`patient_age /= unknown
          OR call_taker_info(st)`n_patients = 1}):
        proQA_system_state
 = st WITH [ call_taker_info := call_taker_info(st)
            WITH [ patient_gender := gen ] ]
```

The subtype constraint we had to introduce in the specification already suggests possible areas that may require further investigation in the DiCoT analysis: Why did the application designers enforce the mentioned restriction? Is the restriction actually implemented in the software used in the Central Ambulance Control room of the London Ambulance Service? If so, what is the actual procedure followed by call-takers when the information system reaches that state? The exercise alone of formalising this activity from the user manual can therefore be used to catch the investigator's attention about potential issues that are latent in the system (in this case, a situation where call-takers are not able to proceed), and stimulate specific questions that could be asked during the field study including finding out whether there are any important ramifications around the issue identified.

In the following, we show how PVS can provide further support by mechanically checking whether the two developed specifications provide a coherent description of the propagation of incident details from call takers to allocators. We aim to check here if any situation may exist where a specification indicates that the allocator has the emergency location and the incident details entered by the call-taker, while the other specification indicates the opposite.

We illustrate how to specify and verify the conjecture when considering the emergency location and the incident details fields. For the emergency location, we describe step-by-step the analysis by using the generic theories developed in section 5.1; rather than, for the incident details we will only discuss the results, as the specification and verification procedure are similar to those used for the emergency location field. For the illustrative purposes of the example, we will consider only two transitions relations, `ct_enters_details` (from the specification of the DiCoT model) and `proQA_incident_details` (from the specification of the user manual). The same approach can be used with any other corresponding transition functions.

The first step for developing the PVS specification for checking the property is to define a data-type for the abstract state (see the procedure explained in section 6.2). The abstract state reflects the safety concerns deemed relevant for the situation. In this case, a suitable abstract state encodes the status of information available to call takers and allocators. We are only interested in checking whether information is available, rather than the actual content. The abstract state is therefore a record with two Boolean fields: `call_taker_has_location`, which is *true* when the call-taker enters information about the location in the system, and `allocator_has_details`, which is *true* when the allocator is able to view information about the location:

```
safety_state: TYPE
  = [# call_taker_has_location: boolean,
       allocator_has_location : boolean #]
```

The second step is to define the abstraction functions for mapping actual practice and prescribed practice states into abstract domain states. In the specification of actual practice, call takers and allocators enter/can view the location of the incident when field `location` of the incident state is different from `NA`. Therefore, we can define an abstraction function `alpha1` that specifies the transformation by using such conditions. In the function specification, we can conveniently use an uninterpreted constants (`the_incident`) for modelling a generic incident.

```
the_incident: below(MAX_INCIDENTS)
alpha1(st: system_state): safety_state =
 (# call_taker_has_location
      := call_taker_info(st)
           `incident_info`location /= NA ,
     allocator_has_location
      := allocator_info(st)
           `incidents(the_incident)`location /= NA #)
```

In the prescribed practice, call takers enter the location when field `em_address` is different from `NA`; allocators can view the location when the system state contains the CAD number associated with the incident and the associated address is different from `NA`. Therefore, as for the other abstraction function, we can define here `alpha2`, which specifies the transformation by using such conditions. In the specification, we can conveniently use the same uninterpreted constant for the incident (`the_incident`), and define an additional uninterpreted conversion function (`cad`) for associating CAD numbers to incidents.

```
cad(x: below(MAX_INCIDENTS)): below(MAX_CAD)
alpha2(st: proQA_system_state): safety_state =
(# call_taker_has_location
     := call_taker_info(st)`em_address /= NA,
    allocator_has_location
     := member(cad(the_incident),
           allocator_info(st)`cad_numbers)
         AND allocator_info(st)
          `em_addresses(cad(the_incident)) /= NA #)
```

Given the above definitions, we can now specify the conjecture by importing the generic theory `safe_divergence_th`. In the following, we show the specification when considering transitions `ct_enters_details` (form the specification of the DiCoT model) and `proQA_incident_details` (form the specification of the user manual).

```
IMPORTING safe_divergence_th
[ safety_state, system_state,
  proQA_system_state, alpha1, alpha2 ]

 actual_st: VAR system_state
 prescr_st: VAR proQA_system_state
 details  : VAR string

 always_verified_location: CONJECTURE
    safe_divergence?(actual_st, prescr_st)
            (ct_enters_details(the_incident),
             proQA_incident_details(details))
```

The conjecture can be verified automatically in seconds with the `grind` strategy provided by the PVS theorem prover.

The conjecture about incident details can be analysed in a similar way. Interestingly, if we try to verify the conjecture for the same pair of transition relations for the incident details, the PVS theorem prover fails to verify the property. In particular, the theorem prover stops in the following situation: the allocator has the incident details according to the actual practice, but the prescribed practice indicates that the allocator doesn't. On checking the developed specification we see that allocators, according to the user manual, receive the incident details only after a specific activity has been performed by call takers, rather than continuously. This reflect the description provided in the user manual, which suggests that the communication happens only when the ProQA system displays 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 [32]).

Further analysis with PVS also shows that there are other situations where allocators may not receive the incident details. For instance, call takers may delay sending the dispatch code for the incident: *"When appropriate, click on the Delay and Continue button to delay dispatch and continue caller interrogation."* (page 89 of ProQA's user manual [32]). 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 [32]).

Again, the above examples show how a relatively simple use of PVS can pull out important details that may warrant further investigation, highlighting issues for the empirical investigators to query or explicitly observe. When a conjecture cannot be verified, then the proof attempt provides precise insights about why the conjecture does not hold. Even if the issue highlighted here could have been in principle identified manually during the specification process, we note that in the general case the specification of real socio-technical systems can be very large, and therefore having a tool-based support that automates mechanic checks is key to performing a more detailed and rigorous analysis.

6.3 Supporting DiCoT model animation

In the context of this case study, 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 5.3. 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 incident 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 simulation trace is related to the PVS specifications of the DiCoT information flow model. The simulator executes four simulation steps. For simplicity, here we consider a system with two ambulances. We have redefined the `print` function used by the simulator so that it shows only the initial and final states, and the sequence of actions performed.

```
<PVSio> exec(4)(LAS_task(sys)
           (initial_task_status), sys);
== Initial state =============
caller_phone( N/A )
caller_location( N/A )
incident_location( N/A )
incident_details ( N/A )
incident_priority( N/A )
-------------------
incidents( { } )
ambulances( {
 [1] available, at_station
 [2] available, on_street
} )
=================================

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

== Final State  =============
caller_phone( +23 322 3860 843 )
caller_location( valid )
incident_location( Terrace pl. )
incident_details ( N/A )
incident_priority( N/A )
```

```
-------------------
incidents( {
 [1] ( loc(Terrace pl.), det(N/A), prio(N/A) )
} )
ambulances( {
 [1] available, at_station
 [2] available, on_street
} )
================================
```

The textual output generated in this example shows that a simple redefinition of the `print` function of the simulator can be used to tailor the animation, without the need of modifying the developed specification of the DiCoT model. In this case, we generated an output meant to be intelligible by humans. However, the same approach can be also used for generating textual outputs that can be imported by external visualisation tools (in [3], for instance, we used this approach for generating waveforms that could be visualised in a graphical tool).

7 Related work

The idea of using formal methods for the analysis of interactive systems in terms of information resources has been investigated in Wright, Fields and Harrison [13, 41] and Doherty Campos and Harrison [27]. In their approach, they specify the actions carried out by individuals in the system, and how these actions are supported by information resources. Such a specification is then verified with automated reasoning tools for checking whether given user goals are adequately supported. They demonstrate the approach by formalising an interactive control system in Uppaal [22]. Their approach is general and not linked to a specific formalism or tool, and they also argue that a resource-based analysis could be extended to the context of a broader methodology, such as DiCoT [6]. In our work, we explore this possibility. In particular, we show that a fairly simple use of an automated reasoning tool like PVS can help analysts verify properties of interest, and also help field researchers identify issues that may warrant further investigation. In section 5, we also demonstrate that properties of interest can be automatically formulated by the automated reasoning system out of the specifications.

Wright, Fields and Merriam [42] investigated the possibility of defining a conceptual framework for integrating formal methods and empirical approaches for studying interactive systems. They proposed a conceptual framework which integrates formal methods and empirical methods in a cyclical process where the two methods feed each other. They demonstrate the approach with an example based on a web browser. The conceptual framework was applied to the analysis of a remote control system in [15]. This work shares with ours the argument that informal approaches and formal methods have complementary roles in the analysis of the system. Namely, in their works, they argue that extant artefacts and informal understanding of the system can provide insights about usability properties that might be of interest. This informal understanding can then be refined through formal methods by generating design questions and evaluating design alternatives, which can in turn be evaluated empirically, e.g., through prototypes. In our work, we proceed in a similar way: starting from field study data or semi-structured DiCoT models, we specify how information resources are transformed and propagated within the system in higher-order logic, and then mechanically check the logical argument framed in the specifications. To support the analysis, we also formalise user manuals, which provide insights about the designers' point-of-view. As gaps and inconsistencies are uncovered within and between the various specifications, new questions are generated, which can be used to refine the DiCoT analysis. Also, the formal specification can be refined as new facts are discovered — the two methods feed each other.

Tasks and work-flow analysis for checking normative behaviours have been explored in other studies with different techniques and different aims. For instance, in [16], 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 [43] and [14], Petri nets based formalisms are used for modelling and analysing industrial and business processes. The aim of these works is to verify the consistency of the system with respect to prescribed (normative) behaviours. In our work, we extend the analysis by considering a distributed cognition perspective on the system that allows us to consider how the deployment of information resources affects users' tasks. Also, our work is based on an integrated use of formal methods and empirical approaches, and allows us to perform a rigorous analysis of *both* normative behaviours and actual practice.

A systematic analysis of normative behaviour for studying interactive systems has been explored by Bolton et al [9, 10]. In their work, they explicitly consider the environment in addition to the human interactive system. They use a task-based analysis for studying how interactive systems may break down because of unanticipated conditions. In their work, they propose a formal modelling language, denominated Enhanced Operator Functional Model (EOFM), for specifying *normative* human behaviours, i.e., sequences of actions prescribed in user manuals. The task models are then used in combination with a specification of an interactive system for verifying properties of interest in the Symbolic Analysis Laboratory (SAL) [28]. They applied their approach to an example from aviation, where an air-traffic controller has to co-ordinate with the pilots of an aircraft. Formerly, Bolton and Bass [7] applied the approach to the verification of a programmable drug infusion pump, and proposed a framework for modelling the system. This work shares with ours the concerns that (i) the model should take into account the broader system (environment, interactions between individuals and devices, and interactions among individuals), and that (ii) the non-experts of formal methods should be able to use the developed tools for analysing realistic systems. The approach was then extended in [8] to address multi-agent systems, human-human communication, and non-normative behaviour. Our work differs from theirs in that we aim to use formal methods to support informal approaches based on distributed cognition for analysing what users do *in the wild* (i.e., their actual behaviour), which can be different from the normative behaviour (e.g., what is reported in written documents of user manuals). Also, our main concern is not to develop a new modelling language (like EOFM), but to build on the expressiveness of formal specification languages, like typed higher-order logic, to closely resemble informal or semi-formal notations used by non-experts of formal methods. As Wright et al [42] observed, multiple methods and multiple empirical techniques are needed to analyse interactive systems, rather than a single modelling language and environment, as each approach can highlight a different aspect of the system. We follow this philosophy.

The work of Rushby [34] also relates to ours. He uses model checking approaches for comparing plausible mental models developed by users and the actual implementation of the system. He argues that any strong divergence between mental models and device models is a potential cause of “automation surprises”, i.e., situations where the automated system behaves in a way that is different from that expected by the operator. He proposed a constructive method for deriving mental models from the specification of the interactive systems [35], and he applied the approach to the analysis of an MD-88 autopilot system, demonstrating how the model checker could provide precise insights about design aspects that may require further investigation. In our work, we use a similar approach for checking unsafe divergence between actual practice (according to field study data) and prescribed practice (according to user manuals, written protocols, or system implementation). Also, we broaden the system under study as we build on field study data gathered that provides information about the whole physical work-space.

8 Conclusion and future work

We have illustrated how an integrated approach using PVS in a pragmatic way and DiCoT can deliver insights about socio-technical systems in a systematic way. A systematic comparison between actual practice (i.e., what individuals do in the workplace, according to the observations of the field investigator) and normative practice (i.e., what individuals are required to do according, for instance, to written documents or user manuals) proved useful for identifying latent situations that may warrant further investigation.

In many cases, even before using automated reasoning tools and techniques, the formal specification pulled out questions to feed discussions on system design and helped to identify important aspects of the system.

As in [42], we believe that multiple methods and empirical techniques are needed to analyse interactive systems, rather than a single modelling language and environment, as each approach can highlight a different aspect of the system. This is especially true for socio-technical systems, where data collected through field studies represents an essential element for studying the system. From this and other case studies analysed with this combined approach, we have some evidence that formal methods and empirical studies are not alternative approaches for studying a socio-technical system, but instead they complement and refine each other.

The expressiveness of the PVS specification language allowed us to overcome some pre-conceived ideas of field researchers about possible limitations of translating informal descriptions into mathematical specifications. Also, the PVSio extension for animating specifications allowed us to engage with them, even if in a limited way, when checking the correctness of the specification. Though, the way automated reasoning tools, including PVS, are currently packaged is a major barrier when engaging with field researchers. We are exploring ways to mitigate this by developing ad hoc GUIs that allow one to explore simulation traces or generate them interactively through simple push button style interfaces.

We are currently exploring the utility of the approach while a field study is in process. The preliminary results, which are reported in [25], are extremely positive, as the tool is allowing an overall finer-grained analysis by uncovering various latent situations that warranted further investigation.

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. The authors would like to thank Michael Harrison as he facilitated the integration process described in this work.

References

1. Formal specification of the London Ambulance Service in PVS (2012). `http://tinyurl.com/PVS-LAS`
2. Bass, E.J., Feigh, K.M., Gunter, E., Rushby, J.: Formal modeling and analysis for interactive hybrid systems. In: 4th International Workshop on Formal Methods for Interactive Systems (2011)
3. Bernardeschi, C., Cassano, L., Domenici, A., Masci, P.: Debugging PVS specifications of control logics via event-driven simulation. In: Proc. 1st Intl. Conf. on Computational Logics, Algebras, Programming, Tools, and Benchmarking (ComputationTools2010) (2010)
4. Bernardeschi, C., Masci, P., Pfeifer, H.: Early prototyping of wireless sensor network algorithms in pvs. In: M.D. Harrison, M.A. Sujan (eds.) Proc. of SAFECOMP08, *Lecture Notes in Computer Science*, vol. 5219, pp. 346–359. Springer (2008)
5. Bernardeschi, C., Masci, P., Pfeifer, H.: Analysis of wireless sensor network protocols in dynamic scenarios. In: Proc. of SSS09, *Lecture Notes in Computer Science*, vol. 5873, pp. 105–119. Springer (2009)
6. Blandford, A., Furniss, D.: DiCoT: A Methodology for Applying Distributed Cognition to the Design of Teamworking Systems. Interactive Systems pp. 26–38 (2006)
7. Bolton, M.L., Bass, E.J.: Formally verifying human—automation interaction as part of a system model: limitations and tradeoffs. Innovations in Systems and Software Engineering **6**(3), 219–231 (2010). DOI 10.1007/s11334-010-0129-9. URL `http://dx.doi.org/10.1007/s11334-010-0129-9`
8. Bolton, M.L., Bass, E.J Siminiceanu, R.I.: Generating phenotypical erroneous human behavior to evaluate human-automation interaction using model checking. The International Journal of Human-Computer Studies. doi:10.1016/j.ijhcs.2012.05.010. (2012).
9. Bolton, M. L., Bass, E. J., Siminiceanu, R. I.: Using formal verification to evaluate human-automation interaction, a review. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, (in press, 2012)
10. Bolton, M. L., Siminiceanu, R. I., Bass, E. J.: A systematic approach to model checking human-automation interaction using task analytic models. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, 41(5), pp. 961-976. (2011)
11. Butler, R., Sjogren, J.: A PVS Graph Theory Library. NASA Technical Memorandum 1998-206923, NASA Langley Research Center, Hampton, Virginia (1998)
12. Crow, J., Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: Evaluating, testing, and animating PVS specifications. Tech. rep., Computer Science Laboratory, SRI International, Menlo Park, CA (2001)
13. C.Wright, P., Fields, B., Harrison, M.D.: Distributed information resources: A new approach to interaction modelling. In: Proceedings of ECCE8: European Conference on Cognitive Ergonomics, pp. 5–10. EACE (1996)
14. Dun, H., Xu, H., Wang, L.: 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 (2008)
15. Fields, R.: Analysis of erroneour actions in the design of critical systems. Ph.D. thesis, University of York (2001)
16. Foster, H., Uchitel, S., Magee, J., Kramer, J.: An integrated workbench for model-based engineering of service compositions. Services Computing, IEEE Transactions on **3**(2), 131–144 (2010)
17. Furniss, D.: Codifying distributed cognition: A case study of emergency medical dispatch. Master's thesis, UCLIC, UCL Interaction Centre (2004)
18. Furniss, D., Blandford, A.: Understanding emergency medical dispatch in terms of distributed cognition: a case study. Ergonomics Journal **49**, 1174–1203 (2006)
19. Hutchins, E.: Cognition in the Wild, new edn. The MIT Press (1995). URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0262581469`
20. Hutchins, E.: How a Cockpit Remembers Its Speed. Cognitive Science **19**, 265–288 (1995)
21. Kirsh, D., Maglio, P.: On distinguishing epistemic from pragmatic action. Cognitive Science **18**, 513–549 (1994)
22. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. Journal on Software Tools for Technology Transfer **1**, 134–152 (1997)
23. Masci, P., Curzon, P.: Checking user-centred design principles in distributed cognition models: a case study in the healthcare domain. In: USAB 2011: Information Quality in eHealth, 7th Conference of the Austrian Computer Society. Springer Lecture Notes in Computer Science (LNCS) (2011)
24. Masci, P., Curzon, P., Blandford, A., Furniss, D.: Modelling distributed cognition systems in pvs. In: FMIS2011, the 4th Intl. Workshop on Formal Methods for Interactive Systems (2011)
25. Masci, P., Furniss, D., Curzon, P., Harrison, M.D., Blandford, A.: Supporting field investigators with PVS: a case study in the healthcare domain In: SERENE 2012: 4th International Workshop Software Engineering for Resilient Systems, Lecture Notes in Computer Science (LNCS) (2012)
26. Masci, P., Huang, H., Curzon, P., Harrison, M.: Using pvs to investigate incidents through the lens of distributed cognition. In: NASAFM 2012: 4th Nasa Formal Methods Symposium. Springer Lecture Notes in Computer Science (LNCS) (2012)
27. McKnight, J., Doherty, G.: Distributed cognition and mobile healthcare work. In: Proc. of BCS-HCI '08, pp. 35–38. British Computer Society, Swinton, UK (2008)
28. de Moura, L., Owre, S., Ruess, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: R. Alur, D.A. Peled (eds.) Computer Aided Verification: CAV 2004, *Lecture Notes in Computer Science*, vol. 3114, pp. 496–500. Springer-Verlag (2004)
29. Movaghar, A., Meyer, J.: Performability modelling with stochastic activity networks. In: Proc. of the 1984 Real-Time Systems Symposium, pp. 215–224 (1984)
30. Muñoz, C.: Rapid prototyping in PVS. Tech. Rep. NIA Report No. 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA (2003)
31. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: combining specification, proof checking, and model checking. In: R. Alur, T.A. Henzinger (eds.) Computer-Aided Verification, CAV '96, no. 1102 in Lecture Notes

in Computer Science, pp. 411–414. Springer-Verlag, New Brunswick, NJ (1996)

32. Priority Dispatch Corp. Inc.: Proqa 3.4, emergency dispatch software (2005). www.prioritydispatch.net/support/pdf/ProQA_User_Guide.pdf
33. Rajkomar, A., Blandford, A.: Understanding infusion administration in the icu through distributed cognition. Journal of Biomedical Informatics (0), – (2012). DOI 10.1016/j.jbi.2012.02.003. URL http://www.sciencedirect.com/science/article/pii/S153204641200024X
34. Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering and System Safety **75**(2), 167–177 (2002). Available at http://www.csl.sri.com/users/rushby/abstracts/ress02
35. Rushby, J.M.: Modeling the human in human factors. In: SAFECOMP, pp. 86–91 (2001)
36. Shankar, N., Owre, S.: Principles and pragmatics of subtyping in PVS. In: D. Bert, C. Choppy, P. Mosses (eds.) Recent Trends in Algebraic Development Techniques, WADT '99, *Lecture Notes in Computer Science*, vol. 1827, pp. 37–52. Springer-Verlag, Toulouse, France (1999)
37. Sharp, H., Robinson, H., Segal, J., Furniss, D.: 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)
38. Vicente, K.J.: Cognitive Work Analysis : Toward Safe, Productive, and Healthy Computer-Based Work. Lawrence Erlbaum, New Jersey (1999)
39. Werth, J., Furniss, D.: Medical equipment library design: revealing issues and best practice with DiCoT. In: International Health Informatics Symposium (IHI 2012) (2011)
40. Westbrook, J.I., Ampt, A.: Design, application and testing of the work observation method by activity timing (wombat) to measure clinicians' patterns of work and communication. International Journal of Medical Informatics **78** (2009). DOI 10.1016/j.ijmedinf.2008.09.003
41. Wright, P., Fields, B., Harrison, M.: Analyzing human-computer interaction as distributed cognition: the resources model. Human Computer Interaction Journal **15**(1), 1–42 (2000)
42. Wright, P., Fields, B., Merriam, N.: From formal models to empirical evaluation and back again, chap. 13, pp. 283–314. Formal methods in human-computer interaction. Berlin, Springer (1997)
43. Zha, H., van der Aalst, W., Wang, J., Wen, L., Sun, J.: Verifying workflow processes: a transformation-based approach. Software and Systems Modeling pp. 1–12 (2010). 10.1007/s10270-010-0149-9