Science of Computer Programming 111 (2015) 214–247

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Interval-based data refinement: A uniform approach to true concurrency in discrete and real-time systems

Brijesh Dongol[a,*], John Derrick[b]

[a] Department of Computer Science, Brunel University London, UK
[b] Department of Computer Science, The University of Sheffield, UK

ARTICLE INFO

Article history:
Received 4 June 2014
Received in revised form 6 May 2015
Accepted 14 May 2015
Available online 27 May 2015

Keywords:
Refinement
Interval-based reasoning
True concurrency
Discrete time systems
Real-time systems

ABSTRACT

The majority of modern systems exhibit sophisticated concurrent behaviour, where several system components observe and modify the state with fine-grained atomicity. Many systems also exhibit truly concurrent behaviour, where multiple events may occur simultaneously. Data refinement, a correctness criterion to compare an abstract and a concrete implementation, normally admits interleaved models of execution only. In this paper, we present a method of data refinement using a framework that allows one to view a component's evolution over an interval of time, simplifying reasoning about true concurrency. By modifying the type of an interval, our theory may be specialised to cover data refinement of both discrete and real-time systems. We develop a sound interval-based forward simulation rule that enables decomposition of data refinement proofs, and apply this rule to verify data refinement for two examples: a simple concurrent program and a more in-depth real-time controller.

1. Introduction

Data refinement allows one to develop systems in a stepwise manner, enabling an abstract system to be incrementally replaced by a concrete implementation by guaranteeing that every observable behaviour of the concrete system is a possible observable behaviour of the abstract system. A benefit of such developments is the ability to reason at a level of abstraction suitable for the current stage of development, and the ability to introduce additional detail to a system via correctness-preserving transformations. A *representation relation* between concrete and abstract states is often used to link the internal states of the concrete and abstract systems. This enables the state representation at different levels of abstraction to differ. For example, a queue data type may be represented by a sequence in an abstract system and by a linked list in the corresponding concrete implementation, and hence, operations at the abstract level access and modify the sequence, whereas at the concrete level operations access and modify the linked list.

Over the years, numerous techniques for verifying data refinement have been developed for a number of application domains [45], including methods for refinement of concurrent [12] and real-time [23] systems. These methods use frameworks that formalise the behaviour of system components in the traditional manner, i.e., as relations between a pre and post state. Therefore, the refinement relations that are used to verify data refinement are also relations between an abstract and a concrete state.

* Corresponding author.
E-mail address: Brijesh.Dongol@brunel.ac.uk (B. Dongol).

http://dx.doi.org/10.1016/j.scico.2015.05.005
0167-6423/© 2015 Elsevier B.V. All rights reserved.

$AInit$: $\neg grd$

Process ap	Process aq
ap_1: **if** grd **then**	aq_1: **if** b **then**
ap_2: $m := 1$	aq_2: $grd := true$
ap_3: **else** $m := 2$ **fi**	aq_3: **else skip fi**

Fig. 1. Abstract program with guard grd.

In the context of true concurrency, pre/post-state relational models lack the expressive power to reason about simultaneous accesses and modifications to a system's state [48] as they inherently admit an interleaved execution semantics. Thus, one must perform an additional step of reasoning to prove that the true concurrency semantics is indeed captured by the interleaved semantics. In some instances, e.g., real-time systems, the pre/post-state relational model cannot be used to formalise *transient properties* [19,17], which are properties that only hold for a small instant of time, making them physically impossible to detect. Further difficulties arise for relational models when admitting real-world delays, where tolerances required of an implementation are difficult to record abstractly.

We aim to enable reasoning about the *evolution* of a system over its interval of execution [2,42], which may comprise several system states. To this end, we use a framework of *interval predicates* [15,19], which is inspired by both Interval Temporal Logic [37] and Duration Calculus [50]. Notable in our logic is that it incorporates reasoning about *apparent states evaluation* [28,19], which allows one to take into account the low-level non-determinism of expression evaluation at a higher level of abstraction. This makes it possible to model both fine-grained interleaving (in the case of concurrent programs) and sampling errors (in the case of real-time systems). Interval predicates have been used to reason about both discrete-time programs [21,15] and real-time systems [19], however, there does not exist any native support for interval-based data refinement. The methods in [21,15,19] only cope with refinements where the concrete state space is a subset of the abstract.

The main contribution of this paper is an interval-based approach for verifying data refinement, which provides a logic for reasoning about refinement in the presence of true concurrency. Our framework is general in the sense that it presents uniform techniques to reason about both discrete-time and real-time systems — the type of reasoning to be performed can be specialised via different instantiations for the type of an interval. We develop a forward simulation rule for verifying data refinement, and present methods for decomposing proof obligations over common programming constructs. These are applied to verify data refinement of a simple concurrent program and a more complex real-time multipump system. For the real-time example, we incorporate the theory of time bands [9,10], which simplifies reasoning about systems over multiple time granularities. Ours is the first method (to the best of our knowledge) to incorporate data refinement and time bands in system development.

This paper extends [13] by including additional explanations, and the real-time example is new to this paper. At a technical level, the definition of refinement has been improved from [13] to better integrate interval-based reasoning; the theory in [13] contained a mix of state and interval-based reasoning, which complicated parts of the logic. These issues have now been streamlined, allowing our theory and associated proofs to become more concise. The underlying notion of refinement is however unaltered from the notions in [13,45]; namely, a concrete system refines an abstract system if, and only if, every observable behaviour of the concrete is a possible observable behaviour of the abstract.

Motivation and background material for the paper is presented in Section 2, clarifying our notions of state-based data refinement. Our interval-based refinement theory is presented in Section 3, and a methods for decomposing refinement proofs via simulation are presented in Section 4. Section 5 presents different methods for evaluation state predicates over intervals and provides background for our two examples. Methods for reasoning about fine-grained concurrency and a proof of our running example is presented in Section 6. A more complex refinement of a real-time multipump system is given in Section 7.

2. State-based data refinement

In this section, we present motivation for our interval-based model by reviewing data refinement for concurrent programs modelled in a framework of pre/post state relations [44,45]. In particular, we describe some of the commonly occurring difficulties when verifying refinement using forward simulation.

As a running example we consider the abstract program in Fig. 1, written in the style of Feijen and van Gasteren [22], which consists of variables $grd, b \in \mathbb{B}$, $m \in \mathbb{N}$, initialisation $AInit$ and processes ap and aq. Process ap is a sequential program with labels ap_1, ap_2, and ap_3 that tests whether grd holds (atomically), then executes $m := 1$ if grd evaluates to $true$ and $m := 2$ otherwise. Process aq is similar. The program executes by initialising as specified by $AInit$, and then executing ap and aq concurrently by interleaving their atomic statements.

A *state* over $V \subseteq Var$ is of type $\Sigma_V \mathrel{\widehat{=}} V \to Val$, where Var is the type of a variable and Val is the generic type of a value, i.e., are mappings from variables to values. *Program counters* for each process are assumed to be implicitly included in each state to formalise the control flow of a program, e.g., the program in Fig. 1 uses two program counters pc_{ap} and pc_{aq}, where $pc_{ap} = ap_1$ is assumed to hold whenever control of process ap is at ap_1, i.e., if $pc_{ap} = ap_1$, then the next statement that ap will execute is the statement labelled ap_1. After execution of ap_1, the value of pc_{ap} is updated so that either $pc_{ap} = ap_2$ or $pc_{ap} = ap_3$ holds, depending on the outcome of the evaluation of grd.