Research Article

Formal Development of System of Systems

Maryam Kamali,^{1,2} Mats Neovius,¹ Luigia Petre,¹ and Petter Sandvik^{1,2}

¹ *Department of Information Technologies, Åbo Akademi University, Joukahaisenkatu 3-5A, 20520 Turku, Finland*

² *TUCS-Turku Centre for Computer Science, Joukahaisenkatu 3-5A, 20520 Turku, Finland*

Correspondence should be addressed to Mats Neovius; mneovius@abo.fi

Received 30 June 2013; Accepted 28 July 2013

Academic Editors: P. Ciancarini and X. He

Copyright © 2013 Maryam Kamali et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Characterising for contemporary systems is their dependence on constituent systems to provide information, functionality, and scalability. Moreover, as the tasks that systems perform are ever more intimate and critical in their nature, reliability and correctness are great concerns. On these matters, we outline a methodology for formal integration of systems. We claim this formal approach to assist in managing the complexity and correctness, in preserving reliability and in respecting the independence of the constituent systems. As a proof of concept, we integrate two in-house control systems specified independently in the Event-B language with the Rodin Platform tool. Moreover, we show how to introduce a new functionality that is only possible due to the integration. Hence, we formally construct a system of systems and provide the methods for hierarchical integration of those.

1. Introduction

Traditionally applications were developed to run on a certain hardware deployed in a specific environment in order to perform some tasks. This approach is still true for many modern deployments, especially those classified as critical systems having legislative regulations as a common nominator. Regulations on such systems are typically enforced by third-party actors. A third-party actor could be an authority supervising that extra-system interference is considered to an acceptable extent, for example, maintenance interval of the mass transport system's platform screen doors or the calibration interval and tolerance of radars controlling the airspace. Commonly these regulated systems are monoliths; that is, even though the system may rely on constituent sub-systems, all systems are modelled to completion. They are also the kind of implementations for which formal methods have been developed, of algorithmic nature. However, apart from critical systems and monolithic system design, the technological advances have enabled an abundance of deployment opportunities for computational devices. Applications relying on such systems provide a sense of comfort and ease our life making the supporting systems an integral part of our everyday lives.

Prominent properties for such contemporary systems are their independence as well as their distributed nature. The independence properties are due to that such a system performs a task of its own regardless of its environment; for example, a weather forecast system forecasts the weather regardless how, if at all, this information is used. Distribution of systems is motivated by, among others, load balancing, optimization, and physical location. Together, this enables what has come to be known as the information revolution. The vast amount of available information from such systems comes with a tradeoff with respect to controllability and reliability, especially when a fault may propagate hierarchically. Thus, a system relying on such independent constituent systems hiding details may merely integrate them and not specify them.

In this paper, we formally address integration of independent systems to form a new system. More precisely, we integrate two disjoint control systems hiding details to a new system. This requires a method for reliable and systematic integration of the visible parts of the independent constituent systems. On this new system of integrated systems, we show how to formally introduce new functionality. With this, we highlight formal methods as a means to provide a structured and reliable means for integration on top of which new

FIGURE 1: A control system.

features may rigorously be added. These formally introduced new features motivate the system of systems (SoS) approach as the system emerges. The relation of SoS with respect to a system as used in this paper follows that of Boardman and Sauser [1]. Thus, we will provide an answer for the question: how to formally integrate independent systems to a coherent system and how to add functionality in a provably correct manner to this newly created system? Realistically, this is the case with practically all componentised and ubiquitous systems, where an independent component/system is used by some other components/systems. The approach also assists in encapsulating critical parts of the software on top of which features increasing usability and user experience may be provided. Hence, our contribution is in providing a method for integration of systems preserving their independence. This contribution effectively enables formal development of SoS. The contribution is supported by references to the relevant literature and exemplified by an easy to follow case study.

This paper is organised as follows. Section 2 motivates the formal approach and use of simple control systems as an example for managing integration of systems. The section also considers available formal methods and briefly introduced the state based formal method of Event-B including supported features used in this paper. Section 3 further motivates our approach and introduces central characteristics for componentisation and SoS. The section also identifies some challenges that need to be dealt with. Following this, Section 4 briefly introduces the control systems and how these may be integrated (parallel composition) in Event-B. We stress that though Event-B requires composing the control systems to a monolithic structure, this is because of language constraints, and the independence of the constituent systems is preserved. On this, Section 5 defines how such an integrated system may be refined to meet the characteristics of an SoS, that is, to add some functionality that is only possible due to the existence of its parts. Hence, the contribution of this paper is in Sections 4.3 and 5, which together outline a method for reliable integration of system whilst respecting their independence (invariants). Section 6 imposes a critical discussion on this and introduces some related work in addition to related work introduced and discussed in text. Finally, Section 7 concludes this paper.

2. Background and Formal Preliminaries

Software is any meaningful digitally encoded and stored sequence of bits. A finite set of sequentially ordered mathematical logical instructions, called an application, define the transformation of software to a new piece of software. Hence, an application is software of a special kind, engineered with the “purpose to provide a user means to perform a task” [2, 3]. The task is something in the interest of a stakeholder that the computational device may perform by executing an application. Inputs to an application are necessary derived from the implicit environment, encoded explicitly (formally), and defined by some semantics. In other words, the motivation indicating the beginning and end of each task is informal [4]. Hence, it is fair to claim that the formal mode merely extends the informal, and it does not replace it [5]. Whether or not the outcome gives rise to any further action is a matter of decision. Moreover, the inclusion of the informal environment breaks down the purely algorithmic model of the formal mode demanded to show mathematical correctness [6, 7].

What we described above is a control system where the application constitutes the computational part. A control system consists of a nonempty set of sensors that translate real life phenomenon to software (a measure), a controller that consumes software and produce software (an application) and some actuator that consumer software and may or may not produce the action in accordance with the designated task [8, 9], illustrated in Figure 1. Here, the controller decides on the type of action by software, and the actuator merely adjusts something, for instance, a valve, accordingly.

Typically the control systems of critical tasks are closed; that is, the impact of the unpredictable environment to the system is minimised. Obviously, this facilitates controllability that, with respect to the above description, reflects assuming any unpredictability of the environment for the sake of mathematical correctness. Hence, what is possible is showing properties in an idealised world, the *model*. This provides a good foundation to formally specify and prove properties and to apply the correct-by-construction idea. However, this raises concerns on the model’s relatedness to the real deployment environment.

Occasionally, a parallel is drawn between this formal specification and a blueprint of some other engineering disciplines, such as when an architect engineers a blueprint of a bridge [10]. This view is motivated when assuming that the abstract model captures relevant limitations imposed by the deployment environment; like the bridge architect takes into account estimated environmental and traffic conditions of the deployment. However, for SoS predicting the independent constituent systems’ behaviour is a utopia. In SoS a control system alike that in Figure 1 would qualify as a first tier system encapsulating a set of components to a system that over time evolves to a complex construct [11]. The complexity is a result of the number of interconnections, protocols, and interfaces. A classic example of a working SoS is the Internet.

However, these features of unpredictability, reliance on independent systems, and complexity indicate that SoS long for a formal design and development methodology. In a

SoS where a system in tier n may consume the output of a constituent system in tier $n - 1$, the formal methodology should support a means to treat encapsulation and hiding. This encapsulation and hiding, sometimes called abstraction, enable managing complexity in a structured and precise manner. However, with respect to traditional formal design applying the correct-by-construction and top-down refinement idea, the direction is the opposite. That is, instead of an abstract specification that is decomposed and finally adding the physics to this, the theorist's way, we will compose a set of systems to a more abstract system of system specification hiding the details of its constituent systems.

Doing this on high level in a systematic manner requires a formal semantics supporting composition/integration and encapsulation. This is indirectly acknowledged by Selberg and Austin [11], who note that an SoS begins from a single system but note that formal decomposition is demanded. We take the engineer's view and note that rather than decomposition a methodology for composition is needed. Fortunately, the semantics of some formal methods already support such parallel composition. Moreover, the semantics of some formal methods support a notion called superposition refinement [12] by which new features (or new systems) may be added to the system of system. Thus, we will provide a methodology to develop a system in the engineer's way, rather than the theorist's way.

2.1. Formal Methods for Specifying Applications. The motivation for using formal methods in system development is the desire to rigorously analyse a system. For this, it is enough to specify *what* is guaranteed, not *how*, *where*, or *when* this is done. Moreover, this requires expressing the specification in a language associated with a formal semantics, the formal method.

The formal methods are abundant and can roughly be divided into three groups: *event based* including CSP [13], CCS [14], π -calculus [15], and REO [16]; *state based* including Action Systems [17], B [18], Event-B [19], Unity [20], Hoare Triples [21], and wp-semantics [22]; and *property based* typically focusing on the temporalities including LTL [23], CTL [24]. The formal methods chosen for specifying a system is usually dictated by its applicability on the problem setting, its tool support and its features.

According to Selberg and Austin [11], formal methods for SoS do not exist. In favour of this claim we note that an SoS is distributed in its nature and each system has managerial independence [25–27], more on the definition in Section 3. Moreover, as an SoS is constructed to execute where special interest is in the state of the system, over channels or temporalities, we believe that state-based formal methods qualify best. For the state-based specification it is, in some cases, enough to know the interfacing variables values of constituent systems.

As control systems heavily rely on the state of the system, for tool support and support of refinement, we chose the state-based Event-B formal method. Event-B, though being relatively new, is related to Action Systems developed in the early 1980s [17]. Event-B language and the Rodin Platform

FIGURE 2: A machine M and a context C .

tool are described in Section 2.2, whereas refinement is described in Section 2.3.

2.2. The Event-B Language and the Rodin Platform. Event-B is a language for state-based formal specification whose state space is bound by its variables' intervals and its constants. The current state of the system is defined by the momentarily values of the declared variables and constants. We call these together the system variables. The values of these system variables are subject to updates by the so-called *events*. Each event is a guarded atomic set of concurrent variable updates whose execution is atomic. A *guard* of an event is a predicate on the system variables that when becomes true enables the transitional part of an event for execution, often called the event's *action*. These actions support nondeterministic assignment of variables; that is, after a transition a system is in a certain state. When the guards of some event evaluate to true (we say events are *enabled*), one is chosen nondeterministically (demonically) for execution. On a specification level, the execution stops when no event is enabled, as the execution may not proceed. Whether or not this condition is desired is task specific; that is, it can model a deadlock or a desired termination condition.

Assurance of properties on the specification is typically instrumented by invariants. Each invariant is a predicate that needs to hold initially and after execution of each event. Thus, correctness of a specification with respect to its requirements is subject to proving the integrity of the invariants. The semantics of the language is described using *before-after* (BA) predicates [10, 19].

A specification in the Event-B language is two-folded, consisting of a *context* (C) and a *machine* (M); see Figure 2. The context defines the static part of the specification by declaring *constants* and *carrier sets* whose properties are *axiomised* for the specification. A machine optionally *sees* context(s), in which case the machine may use the constants and sets. The machine defines the variables, invariants, and the events of the specification. Hence, the machine declares the events whose preservation of the invariants implies the specification consistency, that is, its correctness. Effectively this means that each invariant (predicate) needs to evaluate true in every reachable state.

The Event-B formal method has tool support in form of the Rodin Platform tool [19, 28]. The Rodin Platform tool employs a proof manager that automatically generates proof obligations. Proof obligations are statements that need

to be proved true for the specification to be correct. These proof obligations are logical sequents that have to do with the nonviolation of the invariants and the refinement relation between specifications. Each proof obligation may either be automatically discharged by the integrated theorem provers or may require interaction from the developer.

2.3. System Development in Event-B. A specification in Event-B may consist of a *context* (C) and a *machine* (M) components. The relation between these is defined in the Rodin Platform tool to be the “sees” where the dynamic M may see the static C . Thus, either the machine or the context is a specification in its own right. However, typically both are used as each dynamic M is supposed to function within some static constraints, the context C . Together these components outline a more realistic specification.

In Event-B a machine is a monolithic structure. Thus, there is no function for parallel composition alike that in Action Systems [29], denoted $\parallel$. However, the Rodin Platform tool supports modularization [30] and decomposition by plugins [31]. Decomposition of a machine M to M_1 and M_2 is implemented by either sharing variable(s) [32] or event(s) [33], originally brought forth by Sere [34], later elaborated by Back and Sere [29], in the Action Systems framework. However, due to the monolithic structure of an Event-B specification, the decomposed “machines” will be specified within one abstract machine and thus, only decomposed “virtually.” From a functional point of view, the complement of decomposition is the parallel composition, that due to the aforementioned reason is missing in Event-B.

In Action Systems, parallel composition composes two disjoint systems to one. This composition is straightforward when there are no overlapping variable names. In case of overlapping names, simple renaming solves this. Thus, applying this in Event-B would be much alike ($M_1 \text{ sees } C_1$) $\parallel$ ($M_2 \text{ sees } C_2$), that is, copying the specification of a disjoint specification into the corresponding part of the other specification. With respect to semantics, this is superposition refinement, discussed below.

In addition to parallel composition and decomposition, the components of the Event-B language are subject to refinement. Refinement that denoted $\sqsubseteq$ is a relation that provides a means for a stepwise introduction of details to a component. The goal with such a development is to reach a specification at a level of concretisation sufficient to define *how* something may be implemented. Sometimes refinement is referred to as stepwise development that enables correctness-by-construction. Two types of refinement exist: vertical, known as data refinement [35], and horizontal, known as superposition refinement [12, 36]. Coarsely, data refinement reduces nondeterminism, strengthens the guard, or defines more precise variables requiring a gluing invariant, whilst superposition refinement introduces new variables and events. The refinement relations are associated with conditions that need to be proved. Moreover, refinements are monotone. This is central assuring that a specification refining only one part is a refinement on the whole; that is, if $M \sqsubseteq M'$, then trivially ($M \text{ sees } C$) $\sqsubseteq$ ($M' \text{ sees } C$). Moreover, superposition refinement supports parallel composition

indirectly; that is, if $M_{\text{main}} = (M_1 \text{ sees } C_1)$ and $M'_{\text{main}} = (M_1 \text{ sees } C_1) \parallel (M_2 \text{ sees } C_2)$, then $M_{\text{main}} \sqsubseteq M'_{\text{main}}$ is a valid refinement whenever there are no overlapping variables. Again, in case of overlapping variables, renaming is sufficient. Moreover, obviously for M'_{main} a valid decomposition is ($M_1 \text{ sees } C_1$) and ($M_2 \text{ sees } C_2$).

3. Integration Methodology and Motivation

For complex systems, dividing the complexity to smaller manageable parts is central. In Event-B this is acknowledged, and plugins to the Rodin Platform tool for splitting a monolithic specification into parts have been developed. These include the Decomposition plug-in [31] and the Modularisation plug-in [30]. They both operate when a system is developed “top-down,” that is, from an abstract model towards a more concrete model. The concern we raise is that when doing this, the abstract model necessarily encompasses all restrictions on its constituent systems. As this is the theorist’s way, we propose a method by which systems off the shelf may be composed, that is, a top-down view on system/component specific development but a bottom-up on integration of systems/components.

A component is defined to be a “unit of composition with contractually specified interfaces and explicit context dependencies only” that is “deployed independently and is subject to composition by third parties” [37, 38]. Key characteristics for information derivation from components include the following [39, 40]:

- (i) *explicit dependencies* that specify the information the application requires in order to provide for its task,
- (ii) *contract and interfaces* specifying the functional and nonfunctional characteristics of the component, that is, what is needed and what is guaranteed typically with pre- and postconditions,
- (iii) *unit of deployment* meaning that the component is an autonomous element that may interact with other components through its interface, and
- (iv) *third-party composability* stating that the component may be further composed.

Moreover, a component has no externally observable state [37, 38] and is used through its interface(s). For the characteristics of a component, (i) and (ii) are trivially considered in an Event-B specification; that is, the dependencies are captured by variable read/writes and contract by the (BA) predicates [10, 19]. Unit of deployment (iii) and third-party components (iv) are guaranteed by a methodology for encapsulation while integrating.

These characteristics of a component underline the approach of “what” over “how” and the motivation of integration in the first place. It also motivates for formal treatment of a system that consists of interacting components. A system is thus an abstract component, whose characteristics are outlined as follows [25–27]:

- (a) *operational independence of the individual systems*; that is, that system performs some useful operations independently;
- (b) *managerial independence of the systems* meaning that the systems generally operate independently. These systems are individually acquired and integrated to the SoS while maintaining an independent and continuing operational existence;
- (c) *geographic distribution* of the systems that are integrated to the SoS that only exchange information and knowledge with one another;
- (d) *emergent behaviour* indicating that the composition of systems is more than the sum of its subsystems. Hence, the SoS has a behaviour of its own that is not present in any of its subsystems;
- (e) *evolutionary development* highlighting that an SoS is never fully formed or complete.

Of these, characteristic (d) is central providing the motivation for the approach, also acknowledge by Selberg and Austin [11]. Hence, with respect to components, this paper considers that a system is a higher level construct that may be composed to an SoS.

These characteristics on components and systems requires considering the following challenges in SoS: (1) the integration to an SoS, (2) restrictions on the integrated systems, and (3) terms of use of the shared variables. This paper will focus on presenting a methodology for (1) integrating independently developed systems in structured manner. We assume the integrated systems to be independent and relying on disjoint components, that is, not to share any sensors or actuators. For this, the temperature and light control systems presented in Section 4 constitute a realistic case study. The other possibilities of shared sensors and shared sensors + actuators are shortly discussed in Section 6.

Having independent systems, we pose only the restriction (2) that the interfaces (variables names and types) must remain unchanged throughout SoS evolution. Thus, (iii) unit of deployment and (iv) third-party composability remain, supported by characteristics (a) and (b). This is supported by monotonicity of refinement of which a component/system at any level may be refined in independence. Again, possible overlapping variable names in the interface are dealt with by renaming. For the use of these variables, the rights (3) of the integrating system must explicitly be declared.

It is well known that read-only access is easily dealt with. However, if writing a shared variable, the correctness of both systems must be preserved. This feature is assured in Event-B indirectly as of the monolithic nature, however, compromising (iii), (b), and (d). We will detail this in Section 5 and discuss it further, with respect to formal system development on a general level in Section 6.

4. Integration of Two Control Systems

The integration of multiple independent specifications to one *main* specification is straightforward. We denote this $\parallel$ and note that it corresponds to the Action Systems parallel

FIGURE 3: Composition of two disjoint control systems.

composition. In the following, we will outline two control systems of an intelligent residential house, a temperature control system *tcs*, and a lighting control system *lcs*, whose formal integration is then shown in Section 4.3, depicted in Figure 3.

4.1. Temperature Control System. We model the temperature control system independently of any other control systems and in a stepwise manner, beginning from an abstract specification. The complete specification is published as a technical report [41], whereas this section only outlines the behavioural part. The sole purpose of the temperature control system is to maintain the temperature inside a house, modelled as a variable that must stay within certain thresholds. For controlling the temperature we model two actuators, one representing air conditioning and one representing heating. These actuators decrease and increase the temperature, respectively, and we also assume that the temperature may change randomly with a granularity of 1°C .

In the initial model we model the temperature thresholds and the events that change them, as well as the states of the actuators and events for switching those states. There is also an abstract event representing change of temperature, which we in the first refinement step split into three new events. The first one of these increases temperature under the condition that heating is turned on, the second lowers temperature under the condition that the air conditioning is turned on, and the third changes the temperature nondeterministically as before, but under the condition that neither one of the actuators is active. In the second refinement of our model, we strengthen the guards of the events turning the actuators on by specifying that the air conditioning only turns on when the inside temperature is at or above its upper threshold and the heating only turns on at or below the lower temperature threshold.

After having modelled the environment, we focus on introducing adaptability into our model. We start with the realistic situation that cooling does not stop immediately after the air conditioning is turned on and dually for heating. For this purpose we introduce variables that state how much

before we reach our desired temperature; the air conditioning or heating should be turned off, respectively, as well as a variable representing the desired temperature and an event allowing the desired temperature to be changed. In this refinement step we also strengthen the guards in such a way that the air conditioning can only turn on when heating is turned off, and vice versa. Continuing the adaptability, in the fourth refinement we introduce the “vacation” mode, represented by a Boolean variable *vacation*. When this mode is turned on, that is, when the occupants of the house are on vacation, the temperature variation is allowed to be quite large, as in the previous refinement step. However, when the house is occupied and more comfort is desired, a more restricted temperature range can be applied. We model this by introducing new threshold for the comfortable temperature and refining the events for turning the actuators on into separate events for when vacation mode is on, using the previous thresholds, and when vacation mode is off, using the newly introduced thresholds for a more comfortable temperature range. We also introduce an event for switching the vacation mode on or off.

4.2. Lighting Control System. We develop the lighting control system stepwise with the intent of having automatic control of the lighting in a house. The development process is described in more detail in our technical report [42], whereas this section only outlines the behavioural part. We start from an abstract specification containing a set of rooms, which in this case represent areas in which the lights are turned on or off together, and events to turn lights on or off in any room. In the first refinement step, we introduce a set of sensors, partitioned into two different types, with each sensor giving as output a certain sensor state. Sensors of the first type we refer to as ambient light sensors, and their sensor state is typically based on the ambient light level being low or high enough for lights being enabled or not. The second type we refer to as motion sensors, and they can be in any of three states: lights on, for instance, when motion is detected; lights unchanged, for instance, when motion is no longer detected; and lights off, for instance, when motion has not been detected for some time. Each room must have at least one of each type of sensor, and each sensor must be associated with at least one room. We refine the previous events for turning lights on or off into separate events that switch the lights or let them remain unchanged, depending on sensor states, and we also introduce an abstract event for changing the state of a sensor. In the second refinement step, we refine this event into specific ones representing the switch into each of the three different states for motion sensors and the two states for ambient light sensors. These are further refined in the third refinement step, where we introduce the notion of time, as a variable representing global time and an event to increment that variable. The idea is that a motion sensor switches into lights off state only after some time has passed since it last detected motion, and for that purpose we introduce variables to keep track of the time each sensor last detected motion.

In the fourth refinement step we make the order in which the events can execute more deterministic, by introducing new variables. For each time cycle, these variables keep track

of checked sensor states and of the rooms in which we have evaluated the status of the lights. We require that all sensors associated with a room must have their states checked before the decision is made whether the lights in a room should be switched or remain unchanged, and that all rooms must have their lighting situation evaluated before time is allowed to advance, thus ensuring that lights will actually turn on or off whenever the situation requires.

4.3. Integration. The two control systems presented in Sections 4.1 and 4.2 are developed top-down. When integrating these, we wish to do bottom-up composition, that is, to compose two control systems to one. For this, we apply parallel composition, as presented in Section 2.3. However, parallel composition is not supported by any tool of Event-B because of the monolithic specification. Thus, we will make use of superposition refinement for the same outcome.

Integrating the two control systems *tcs* and *lcs* to a new system is thus straightforward. Let the new system be called *main*. Then $main = tcs \parallel lcs$ that the absence of parallel composition in Event-B is realised as follows. First we specify *tcs* and *lcs* disjointly. On these disjoint specifications refinement may be applied. At any point in the development with nonoverlapping variable names, *tcs* and *lcs* may be parallel composed by simply refining the one to include the specification of the other, that is, $tcs \sqsubseteq main$ or $lcs \sqsubseteq main$ for the machine part and optional context. The refinement here is superposition or in terms of Event-B *extends*. As of refinement monotonicity, if $tcs \sqsubseteq tcs'$ or if $lcs \sqsubseteq lcs'$, then $main \sqsubseteq main'$. Also the case that $tcs \parallel lcs \sqsubseteq tcs' \parallel lcs'$ implies trivially that $main \sqsubseteq main'$ when there are no shared variables as the invariants of the other are trivially respected [43].

5. Refining the System of System

Assume the *tcs* and *lcs* as in the technical reports [41, 42], respectively, presenting them in detail. Moreover, assume the parallel composition by superposition refinement as presented in Section 4. On this, this section will outline the three possibilities of refining the newly formed system $main = tcs \parallel lcs$ to $main' = tcs \parallel lcs \parallel \langle new\ events \rangle$. For this setting, the *new events* of *main* may interact with interfacing variables of *tcs* and *lcs* in three different ways:

- (i) read-only,
- (ii) read and write with restrictions,
- (iii) read and write.

The difference between these alternatives is that this depends on whether or not *main* encapsulates the interfaces as well. These are depicted in Figure 4 where the arrow head dictates the direction of data flow; that is, here (red) out arrows are values that may be read and (black) in arrows values that are written. In the following, we will overview each scenario.

The simplest case of interaction between the components of *main* is case I, read-only. As noted in Section 3 it is well known and easily dealt with. The key is that by only consuming the result of another component, the interaction

FIGURE 4: System interaction possibilities, cases I, II, and III.

```

event RAISE_ALARM
where
  @grd6.1  vacation = TRUE
  @grd6.2   $\exists s \cdot s \in \text{motionsensors} \wedge \text{sensor\_state}(s) = \text{sensor\_lights\_on}$ 
then
  @act6.1  alarm := TRUE
end

```

ALGORITHM 1: Event RAISE_ALARM.

is one way. However, this is the structure for the purest form of componentisation that conforms to the characteristics of a component as listed in Section 3. The concrete components, here *tcs* and *lcs*, encapsulate the aggregation or composition of directly connected sensors to provide a “composite state”. Hence, *tcs* and *lcs* may define the readable variables. Another upside for this is that the methodology may be applied on services as well, only “using” the underlying components. In the control system setting, this is reasonable as the means of actuation may not happen through the underlying component but is separate or embedded within a component.

This is the case for our specification as well, where *main* is refined with three new events for providing burglar detection. Hence, *<new events>* constitutes events for burglar detection. This specific feature requires a Boolean variable for the state of alarm, called *alarm* that is initially assigned FALSE. The burglar detection feature also requires reading the Boolean variable of vacation of *tcs* that is TRUE if the inhabitants are on vacation and false otherwise. It also reads all motions sensors states from all areas in *lcs*; that is, *sensor_state* $\in \{\text{sensor_lights_unchanged}\}, \{\text{sensor_lights_off}\}, \{\text{sensor_lights_on}\}$ that is of the type **motionsensors**. Obviously, the burglar is detected and event RAISE_ALARM is enabled if *vacation* = TRUE and *sensor_state* for some sensor in any area indicates movement, in case variable *alarm* is assigned TRUE (see Algorithm 1).

We made the design decision that the burglar leaving the area is also captured. Thus, an event STOP_ALARM_MOTION_TIMEOUT is defined assigning variable *alarm* to FALSE (see Algorithm 2).

In addition, the event STOP_ALARM_VACATION_ENDED captures the situation when the vacation of the inhabitants ends, and consequently the variable *alarm* is assigned FALSE (see Algorithm 3).

With this, we have realised formally an SoS out of components in the purest form by parallel composition *main* = *tcs* || *lcs* and then superposition refining *main* to *main'* =

tcs || *lcs* || *<new events>*. Here the components may refine independently.

Reading and writing with restrictions to the interfacing variables include the restriction, case II, of not changing enabledness of any event. This is reasonable in special cases including, for example, situations when the providing system implements a counter that is increased for each read access. This problem has been considered in Action Systems [44]. Implementation of this constitutes a sequence of two actions with one doing the read and related updates and the other writing to a variable in reaction, that is, increasing the value on a counter. Thus, the implementation on a semantic level is identical, if *main* = *tcs* || *lcs* then *main'* = *tcs* || *lcs* || *<new events>*. Each of the new components may be refined independently.

For case III, the options have been studied by Back and Sere [29]. In short, there are two ways to treat this, by either sharing a variable or by sharing an event. This is the same way as decomposition does it, by sharing a variable(s) [32] or event(s) [33]. In Event-B this poses no difficulty, again due to the monolithic structure. However, for componentisation, this compromises the independence. Thus, as depicted in Figure 4, the *main* system encapsulates its used components. Realistically, this is the case when the component is “owned,” that is, if the software is run locally or if a unit of deployment is bought and integrated to some existing system. Hence, the implementation in terms of this paper where *main* = *tcs* || *lcs* is *main'* = *tcs'* || *lcs'* || *<new events>* where *new events* = *<non interacting events>* || *<new shared events>*. Note that here also *tcs* and *lcs* refine to *tcs'* and *lcs'* as of including the new interaction variable. Hence, any refinement here contributes to refining the composed system.

6. Discussion

A formal method as described in this paper is a language with well-formed semantics in which a specification describing

```

event STOP_ALARM_MOTION_TIMEOUT
  where
    @grd6_1  $\forall s \cdot s \in \text{motionsensors} \Rightarrow \text{sensor\_state}(s) = \text{sensor\_lights\_off}$ 
    @grd6_2 alarm = TRUE
  then
    @act6_1 alarm := FALSE
  end

```

ALGORITHM 2: Event STOP_ALARM_MOTION_TIMEOUT.

```

event STOP_ALARM_VACATION_ENDED
  where
    @grd6_1 vacation = FALSE
    @grd6_2 alarm = TRUE
  then
    @act6_1 alarm := FALSE
  end

```

ALGORITHM 3: Event STOP_ALARM_VACATION_ENDED.

“what” holds may be expressed. As of this, it may be argued that a programming language is a formal method. Such an argument would typically be disregarded by the notion that the executable code defines “how,” which obviously implies “what.” Hence, a programming language is incapable of abstraction, whereas a formal method provides a mathematical system based on which we reason on an abstract level. This reasoning typically strives for showing mathematical correctness. However, defining correctness is a serious challenge in engineering [45]. Another paradox with formal methods is that many of them strive for code generation, that is, fuzzyfying the whole concept of a formal method with respect to a skeleton defined in a programming language.

Showing correctness of a specification is possible mainly because the model on the environment simplifying the deployment domain to an extent that correctness may be shown. However, this model or “assumption” of imperfect matters is a cause of faults and failures [45] and enables that exactness from where formal methods derive their strength [46]. On this, capturing the model’s closeness to the real world becomes interesting, an issue noted frequently [10] but not considered in depth.

Thus, the question deviates to what is beneficial with formal methods with respect to a programming language. Often this is, as noted, the possibility to reason rigorously on an abstract level. Indeed, there are some approaches for correctness preserving reasoning [17, 20, 47, 48], several on methods implementing these including those used by Event-B or inspired by it [29, 32, 33, 49], an abundance of academic case studies, and some industrial success stories.

We agree on the benefits of rigorous reasoning on an abstract level made possible by formal methods. However, unlike reasoning on an application level from a theorist’s view, we see great potential in applying formal methods to manage complexity by componentisation. With respect to this, we

introduced in this paper a method for integration of systems to an SoS and showed the possibility for emergent behaviour of a system (characteristic (d)) by superposition refinement of the parallel composed constituent system. Hence, we do not claim that the systems *tcs*, *lcs*, or the *<new_events>* or their composition are complete but acknowledge these to be subject for tuning if implemented. Formally we may guarantee the consistency of the SoS; that is, that no incorrectness is introduced as of the development process. This is ascertained because of the read-only or read-write with restrictions assumption.

As this assumption on interaction may appear odd and restrictive, we note that in Event-B, new events introduced in a refinement step are generally not allowed to modify existing variables. The reason is because this could cause inconsistent behaviour with respect to the previous refinement step. Therefore, if we want to combine two systems in the way we have done, these systems must be truly independent and not share any variables. However, when we have combined our two systems we may use the variables from each system both in new invariants and as guards of refined events regardless the previous systems the events originate from. Thus, when continuing to refine our combined systems, we cannot have events that simultaneously modify two variables previously belonging to separate systems, but we can refine an event from one of the systems to act based on variables from the other. An example of this would be a further refinement, in which the events turning off the heater would act also based on the state of the ambient light sensors; that is, the decision to turn off the heater would be made sooner if it is sunny outside, under the assumption that sunshine would also heat the house.

Related theories developed for reasoning on systems include that of general system theory [50]. The general system theory captures relations on its basic elements. With respect to this, we show in this paper how to integrate systems

rigorously in an acyclic manner while further superposition refining them. Moreover, we allow the integrated systems to preserve privacy by encapsulating their constituent systems and abstracting their functionality.

Although we cannot combine events to fully integrate our previously distinct systems, we can introduce new variables and events that depend on them. We have shown an example of this, in which a burglar alarm is activated based on the vacation mode setting from the temperature control system and the motion sensor states from the lighting control system. Thus, we have introduced functionality that would not be possible without us having integrated our systems into such a system of systems.

In addition, we acknowledge that composing control systems of various kinds may interfere with each other. For example, if we had a humidity control system, then switching AC on in *tcs* would surely affect the humidity in that area. This is a problem of most of the real implementations. However, as this paper focuses on presenting a methodology, that of superposition refinement in Event-B to manage SoS development, we allow us therefore to disregard this.

What comes to the sharing of sensors or sensors + actuators, the sharing of sensors has been considered in terms of context [51]. This is noted to be of no problem whenever the SoS network contains no undirected loops, that is, a polytree. With respect to the actuators, we see no problem of sharing.

7. Conclusions

Technological advances have enabled an abundance of deployment opportunities for computational systems. These contemporary systems are often distributed and independent of each other. By integrating two or more systems we may draw from the features of each system in order to create something that is more than just the sum of its parts. Such a system, composed of smaller systems but with functionality enabled only by integrating features from several systems, is known as an SoS [25–27].

In this paper, we have shown a method of formally integrating disjoint systems, enabling us to create an SoS by adding functionality that depends on both of the previously separate systems. We make use of two control systems [41, 42] independently developed by stepwise refinement using the Event-B language [10] and the Rodin Platform tool [28], enabling us to prove the correctness of each step. Then, we have integrated these control systems and expanded upon them while preserving the correctness of the whole SoS. While there are methods for dividing the complexity of a system in Event-B by splitting it into smaller systems [32, 33, 49], our approach has been different in that we have not started with an abstract specification of what the whole integrated system should achieve, but with the development of two completely independent subsystems. Hence, we have started from a system, as indicated by Selverg and Austin [11]. Our approach has the advantage of not directly imposing all the restrictions from the abstraction of the whole system onto the constituent systems, thereby increasing their reusability compared to a situation where the constituent systems had

been designed from the ground up as parts of a larger system. We have also shown that integrating control systems by this way enables us to easily add functionality that depends on all the constituent systems.

We believe that our approach to formally integrating systems into a SoS can be useful in situations where reusing existing components is more desirable than building a whole system from bottom-up, or where the behaviour of the whole system is not relevant when specifying the behaviour of the constituent systems. By developing constituent systems formally and independently, managing the complexity is facilitated by a structured and rigorous semantics. This supports the expansion of functionality that is motivating for an SoS.

Acknowledgement

This research is funded by Academy of Finland project “FResCo: High-quality Measurement Infrastructure for Future Resilient Control Systems” (Grant number 263925).

References

- [1] J. Boardman and B. Sauser, “System of systems-the meaning of of,” in *Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering*, pp. 118–123, April 2006.
- [2] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski, “Challenges: an application model for pervasive computing,” in *Proceedings of the 6th Annual international Conference on Mobile Computing and Networking (MOBICOM ’00)*, pp. 266–274, August 2000.
- [3] G. Banavar and A. Bernstein, “Software infrastructure and design challenges for ubiquitous computing applications,” *Communications of the ACM*, vol. 45, no. 12, pp. 92–96, 2002.
- [4] H. Zemanek, *Abstract Architecture, General Concepts for Systems Design*, Lecture Notes in Computer Science 86/1980, Winterschool on Abstract Software Specification at the Danish University of Technology, Copenhagen, Denmark, 1980.
- [5] P. Naur, “Formalization in program development,” *BIT Numerical Mathematics*, vol. 22, no. 4, pp. 437–453, 1982.
- [6] M. Shaw, “Beyond objects: a software design paradigm based on process control,” *ACM SIGSOFT Software Engineering Notes*, vol. 20, pp. 27–38, 1995.
- [7] M. Shaw and D. Garlan, *Software Architecture, Mona Pompili*, Prentice-Hall, New Jersey, NJ, USA, 1996.
- [8] G. Biegel and V. Cahill, “A framework for developing mobile, context-aware applications,” in *Proceedings of the 2nd IEEE international Conference on Pervasive Computing and Communications (Percom ’04)*, pp. 361–365, March 2004.
- [9] A. Fitzpatrick, G. Biegel, S. Clarke, and V. Cahill, “Towards a sentient object model,” in *Proceedings of the Workshop on Engineering Context-Aware Object Oriented Systems and Environments (OOPSLA/ECOOSE ’02)*, 2002.
- [10] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, New York, NY, USA, 2010.
- [11] S. Selberg and M. Austin, “Toward an evolutionary system of systems architecture,” in *Proceedings of the 18th Annual International Symposium of The International Council on Systems Engineering (INCOSE ’08)*, 2008.

- [12] R. J. R. Back and K. Sere, "Superposition refinement of reactive systems," *Formal Aspects of Computing*, vol. 8, no. 3, pp. 324–346, 1996.
- [13] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.
- [14] R. Milner, *A Calculus of Communicating Systems*, Springer, New York, NY, USA, 1980.
- [15] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.
- [16] F. Arbab, "Reo: a channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, vol. 14, no. 3, pp. 329–366, 2004.
- [17] R. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," in *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1983.
- [18] J.-R. Abrial, *The B-Book: Assigning programs to meanings*, Cambridge University Press, New York, NY, USA, 1996.
- [19] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 6, pp. 447–466, 2010.
- [20] K. Mani Chandy, *Parallel Program Design: A Foundation*, Addison-Wesley Longman Publishing, Boston, Mass, USA, 1988.
- [21] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, 1969.
- [22] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.
- [23] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pp. 46–57, 1977.
- [24] E. A. Emerson and J. Y. Halpern, "Decision procedures and expressiveness in the temporal logic of branching time," *Journal of Computer and System Sciences*, vol. 30, no. 1, pp. 1–24, 1985.
- [25] M. W. Maier, "Architecting principles for system of systems," *Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.
- [26] A. Sage and S. Biemer, "Processes for system family architecting, design, and integration," *IEEE Systems Journal*, vol. 1, no. 1, pp. 5–16, 2007.
- [27] A. Sage and C. Cuppan, "On the systems engineering and management of systems of systems and federations of systems," *Information Knowledge Systems Management*, vol. 2, no. 4, pp. 325–345, 2001.
- [28] "Event-B and the Rodin Platform," <http://www.event-b.org/>, 2013.
- [29] R. Back and K. Sere, "From action systems to modular systems," in *Proceedings of the 2nd International Symposium of Formal Methods Europe. Industrial Benefit of Formal Methods (FME '94)*, pp. 1–25, 1994.
- [30] "Rodin Modularisation Plug-in," http://wiki.event-b.org/index.php/Modularisation_Plug-in, 2013.
- [31] "Decomposition Plug-in User Guide," http://wiki.event-b.org/index.php/Decomposition_Plug-in_User_Guide, 2013.
- [32] J. R. Abrial, "Event Model Decomposition," Wiki document Version 1. 3, http://wiki.event-b.org/images/Event_Model_Decomposition-1.3.pdf, 2009.
- [33] M. Butler, "Decomposition structures for event-B," in *Proceedings of the 7th International Conference on Integrated Formal Methods (IFM '09)*, 2009.
- [34] K. Sere, *Stepwise derivation of parallel algorithms [Ph.D. thesis]*, Åbo Akademi, Department of Computer Science, Turku, Finland, 1990.
- [35] R. Back and K. Sere, "Stepwise refinement of action systems," *Structured Programming*, vol. 12, no. 1, pp. 17–30, 1991.
- [36] S. Katz, "Superimposition control construct for distributed systems," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 2, pp. 337–356, 1993.
- [37] C. Szyperski, *Component Software*, Addison-Wesley, Reading, Mass, USA, 1998.
- [38] C. Szyperski, D. Gruntz, and S. Murer, *Component Software-Beyond Object-Oriented Programming-Second Edition*, Addison-Wesley and ACM Press, Reading, Mass, USA, 2002.
- [39] T. Broens, *Dynamic context bindings-Infrastructural support for context-aware applications [Ph.D. thesis]*, University of Twente, Enschede, The Netherlands, 2008, CTIT Ph.D.thesis series No. 08-125.
- [40] M. Wegdam, *Dynamic reconfiguration and load distribution in component middleware [Ph.D. thesis]*, University of Twente, Enschede, The Netherlands, 2003.
- [41] M. Neovius, "Formal stepwise development of an in-house temperature control system," TUCS Technical Reports 1078, 2013.
- [42] P. Sandvik, "Formal stepwise development of an in-house lighting control system," TUCS Technical Reports 1079, 2013.
- [43] R. J. R. Back and J. von Wright, "Compositional action system refinement," *Formal Aspects of Computing*, vol. 15, no. 2-3, pp. 103–117, 2003.
- [44] M. Neovius and K. Sere, "Formal modular modelling of context-awareness," in *Proceedings of the 7th International Symposium in Formal Methods for Components and Objects*, Revised Lectures, pp. 102–118, 2008.
- [45] D. L. Parnas, "Really rethinking 'formal methods,'" *Computer*, vol. 43, no. 1, Article ID 5398780, pp. 28–34, 2010.
- [46] H. Lieberman and T. Selker, "Out of context: computer systems that adapt to, and learn from, context," *IBM Systems Journal*, vol. 39, no. 3-4, pp. 617–632, 2000.
- [47] R. Banach and M. Poppleton, "Sharp retrenchment, modulated refinement and simulation," *Formal Aspects of Computing*, vol. 11, no. 5, pp. 498–540, 1999.
- [48] L. Lamport, "Temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, 1994.
- [49] A. Iliasov, E. Troubitsyna, L. Laibinis et al., "Supporting reuse in event B development: modularisation approach," in *Proceedings of Abstract State Machines, Alloy, B, and Z, (ABZ, 2010)*, pp. 174–188, 2010.
- [50] L. von Bertalanffy, *General System Theory: Foundations, Development, Applications*, Allen Lane, London, UK, 1971.
- [51] M. Neovius, *Trustworthy context dependency in ubiquitous systems [Ph.D. thesis]*, Åbo Akademi University, Turku, Finland, 2012, TUCS Dissertations 151.